DOCUMENT RESUME

ED 252 602 TM 850 080

AUTHOR Little, Glenn A.; And Others
TITLE A Computer-Based Gaming System for Assessing Recognition Performance (RECOG).
INSTITUTION Navy Personnel Research and Development Center, San Diego, Calif.
REPORT NO NPRDC-TL-85-3
PUB DATE Jan 85
NOTE 101p.
PUB TYPE Reports - Research/Technical (143) -- Computer Programs (101)

EDRS PRICE MF01/PC05 Plus Postage.
DESCRIPTORS Adults; Computer Graphics; *Computer Simulation; *Computer Software; Games; Perception Tests; *Recognition (Psychology); *Visual Perception; Visual Stimuli

ABSTRACT

This report documents a computer-based gaming system for assessing recognition performance (RECOG). The game management system is programmed in a modular manner to: instruct the student on how to play the game, retrieve and display individual images, keep track of how well individuals play and provide them feedback, and link these components by supervising routines in order to execute the game. This modularity in programming, together with the game management system's independence of any graphic database (e.g., aircraft or ship silhouettes, human anatomy, topography, electronic circuits), contributes to its generalizability. The game, then, provides a set of software tools which can be used by others who want to assess recognition performance. The software for the complete gaming system is currently on three floppy disks which control the play of the game, contain the graphic-images database, and maintain records of individuals' recognition performances. The game itself is run with two dual-density disks on the Terak microcomputer employing two drives. It is implemented on the UCSD P-System and written in UCSD PASCAL. The program listing is included in Appendix A. (Author/BW)

****
* Reproductions supplied by EDRS are the best that can be made *
* from the original document. *
****

ED252602

TL 85-3

JANUARY 1985

A COMPUTER-BASED GAMING SYSTEM FOR ASSESSING RECOGNITION PERFORMANCE (RECOG)

"PERMISSION TO REPRODUCE THIS MATERIAL HAS BEEN GRANTED BY

Navy Personnel R&D Center

TO THE EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)."

U.S. DEPARTMENT OF EDUCATION
NATIONAL INSTITUTE OF EDUCATION
EDUCATIONAL RESOURCES INFORMATION CENTER (ERIC)

X This document has been reproduced as received from the person or organization originating it.
Minor changes have been made to improve reproduction quality.

• Points of view or opinions stated in this document do not necessarily represent official NIE position or policy.

TM 850 080

TRAINING LABORATORY
NAVY PERSONNEL RESEARCH AND DEVELOPMENT CENTER
SAN DIEGO, CALIFORNIA 92152

A COMPUTER-BASED GAMING SYSTEM FOR ASSESSING RECOGNITION PERFORMANCE (RECOG)

Glenn A. Little
Donald H. Maffly
Corbin L. Miller
David A. Setter
University of California, San Diego

Pat-Anthony Federico
Navy Personnel Research and Development Center

Reviewed and approved by
James S. McMichael

Released by
James S. McMichael
Director, Training Laboratory

Training Laboratory
Navy Personnel Research and Development Center
San Diego, California 92152

UNCLASSIFIED

SECURITY CLASSIFICATION OF THIS PAGE

REPORT DOCUMENTATION PAGE

1a. REPORT SECURITY CLASSIFICATION: UNCLASSIFIED	1b. RESTRICTIVE MARKINGS:
2a. SECURITY CLASSIFICATION AUTHORITY:	3. DISTRIBUTION/AVAILABILITY OF REPORT: Approved for public release; distribution unlimited.
2b. DECLASSIFICATION/DOWNGRADING SCHEDULE:	
4. PERFORMING ORGANIZATION REPORT NUMBER(S): NPRDC TL 85-3	5. MONITORING ORGANIZATION REPORT NUMBER(S):
6a. NAME OF PERFORMING ORGANIZATION: Navy Personnel Research and Development Center; 6b. OFFICE SYMBOL (If applicable):	7a. NAME OF MONITORING ORGANIZATION:
6c. ADDRESS (City, State and ZIP Code): San Diego, California 92152	7b. ADDRESS (City, State and ZIP Code):
8a. NAME OF FUNDING/SPONSORING ORGANIZATION: Chief of Naval Material Office of Naval Technology; 8b. OFFICE SYMBOL (If applicable):	9. PROCUREMENT INSTRUMENT IDENTIFICATION NUMBER:
8c. ADDRESS (City, State and ZIP Code): Washington DC 20360	10. SOURCE OF FUNDING NUMBERS: PROGRAM ELEMENT NO. 63720N; PROJECT NO. RF63-522; TASK NO. 801-013; WORK UNIT NO. 03.04

11. TITLE (Include Security Classification): A Computer-Based Gaming System for Assessing Recognition Performance (RECOG)

12. PERSONAL AUTHOR(S): G. A. Little, D. H. Maffly, C. L. Miller, D. M. Setter, & P-A. Federico

13a. TYPE OF REPORT	13b. TIME COVERED	14. DATE OF REPORT (Year, Month, Day)	15. PAGE COUNT
Technical Report	FROM Sep 83 TO Sep 84	January 1985	99

16. SUPPLEMENTARY NOTATION:

17. COSATI CODES			18. SUBJECT TERMS (Continue on reverse if necessary and identify by block number)
FIELD	GROUP	SUB-GROUP	Computer-Based Testing; Assessing Recognition Skills; Computer-Based Game; Testing Software Tools

19. ABSTRACT (Continue on reverse if necessary and identify by block number)

This report documents a computer-based gaming system for assessing recognition performance (RECOG). This was done so that others who may want to use it for either research, development, or operational implementation will have an easier time comprehending the modularity of the programming structure as well as how specific procedures can be adapted to suit a user's unique situation. The game management system is programmed in a modular manner to: instruct the student on how to play the game, retrieve and display individual images, keep track of how well individuals play and provide them feedback, and link these components by supervising routines inorder to execute the game. This modularity in programming, together with the game management system's independence of

20. DISTRIBUTION/AVAILABILITY OF ABSTRACT	21. ABSTRACT SECURITY CLASSIFICATION	
☒ UNCLASSIFIED/UNLIMITED ☐ SAME AS RPT ☐ DTIC USERS	UNCLASSIFIED	
22a. NAME OF RESPONSIBLE INDIVIDUAL: Pat-Anthony Federico	22b. TELEPHONE (Include Area Code): (619) 225-6434	22c. OFFICE SYMBOL: Code 51

DD FORM 1473, 84 JAN — 83 APR EDITION MAY BE USED UNTIL EXHAUSTED ALL OTHER EDITIONS ARE OBSOLETE

UNCLASSIFIED
SECURITY CLASSIFICATION OF THIS PAGE

any graphic database (e.g., aircraft or ship silhouettes, human anatomy, topography, electronic circuits), contributes to its generalizability. The game, then, provides a set of software tools which can be used by others who want to assess recognition performance.

The software for the complete gaming system is currently on three floppy disks which control the play of the game, contain the graphic-images database, and maintain records of individuals' recognition performances. The game itself is run with two dual-density disks on the Terak microcomputer employing two drives. It is implemented on the UCSD P-System and written in UCSD PASCAL. The disk placed in drive 0, i.e., the 8510 or volume 4, holds the actual game code; the disk placed in drive 1, i.e., the 8515 or volume 5, contains the independent graphic-images database. As soon as the system is booted, control is immediately passed to the game. Consequently, naive users need not deal with the nuances of the UCSD P-System. Recognition-performance data are saved for a number of individual players on the 8510 disk drive. A third disk containing game management facilities can be used by test administrators or researchers to format the recognition data to facilitate statistical analyses. Also, this third disk can be used to design a new game with a completely different set of graphic images to act as stimuli for recognition testing.

FOREWORD

This programming effort was performed under exploratory development work unit RF63-522-801-013-03.04 (Testing Strategies for Operational Computer-Based Training) sponsored by the Chief of Naval Material (Office of Naval Technology). The objective of this work unit is to develop and evaluate microcomputer-based graphic simulations of operationally oriented tasks to determine if they result in better assessment of student performance than more customary measurement methods.

This program documentation is primarily intended for the Department of Defense training and testing research and development community.

SUMMARY

Background and Problem

The general goal of this exploratory development is to create and evaluate microcomputer-based graphic simulations of operationally oriented tasks to ascertain if they result in improved assessment of student performance when compared to more customary measurement methods. As a test bed, graphic models have been programmed to assess how well F-14 Pilots and Radar Intercept Officers (RIOs) recognize front-line Soviet and non-Soviet fighters and bombers.

A computer game based upon a sequential recognition paradigm has been designed and developed. It randomly selects and presents on the display of a microcomputer with millisecond speed either the front, side, or top views of four Russian bombers and ten of their advanced fighters. Also, the game management system can choose and flash corresponding silhouettes of NATO aircraft which act as distractors for their Soviet counterparts because of the high degree of similarity between them which could easily confuse U.S. air crews.

This game, which is called FLASH IVAN (aircraft images are "flashed" on the computer display, and the F-14 community refers to the Russians generically as "Ivan"), assesses student performance by measuring the number of correct recognitions out of a total of forty-two silhouettes (half Soviet and the other half non-Soviet), the time it takes a student (latency) to make a recognition judgment for each target or distractor aircraft, and the degree of confidence the student has in each of his/her recognition decisions. At the end of the game feedback is given to the student concerning his percentage of correct recognitions, average response latency, average degree of confidence in the recognition judgments, and how his performance compares to other students who have played the game.

A file is maintained and available to the instructors which provides, in addition to these parameters for each student, recognition performance across aircraft for all students who played the game. This provides diagnostic assessments to instructors who can use this information to focus student attention on learning the salient distinctive features of certain aircraft in order to improve their recognition performance.

The software for the complete gaming system is currently on three floppy disks which control the play of the game, contain the graphic-images database, and maintain records of individuals' recognition performances. The game itself is run with two dual-density disks on the Terak microcomputer employing two drives. It is implemented on the UCSD P-System and written in UCSD PASCAL. The disk placed in drive 0, i.e., the 8510 or volume 4, holds the actual game code; the disk placed in drive 1, i.e., the 8515 or volume 5, contains the independent graphic-images database. As soon as the system is booted, control is immediately passed to the game. Consequently, naive users need not deal with the nuances of the UCSD P-System. Recognition-performance data are saved for a number of individual players on the 8510 disk drive. A third diskette containing game management facilities can be used by test administrators or researchers to format the recognition data to facilitate statistical analyses. Also, this third diskette

can be used to design a new game with a completely different set of graphic images to act as stimuli for recognition testing.

Objective

The objective of this report is to document the program underlying the computer-based gaming system. This was done so that others who may want to use this set of software tools for either research, development, or operational implementation will have an easier time comprehending the modularity of the programming structure as well as how specific procedures can be adapted to suit a user's unique situation.

Utility Functions

This section of the documentation describes how to create new recognition games which would employ as subject-matter databases graphic images other than aircraft silhouettes currently used by Flash Ivan. It also explains how to extract statistical data for sample of subjects from records of recognition performances.

Programmer's Notes

This portion of the documentation serves as a technical reference for programmers who may want to make slight modifications to the game code itself which is independent of the database. It deals with several files and describes procedures which would be involved in performing these changes.

Program Maintenance

The final segment of the documentation explains how to maintain the program, the organization and the handling of the three disks that are used, and what to do to the disks before and after collecting recognition-performance data. A listing of the program is presented in Appendix A.

CONTENTS

Page

INTRODUCTION 1
- Background and Problem 1
- Objective 3

UTILITY FUNCTIONS 4
- The "ADM" Disk 4
- Creating a Game 5
- Creating Game Images 6
- Creating the Image Directory 7
- More Utilities 10
  - Converting the Directory 10
  - Quicklist 11
  - Making Hiscores File 11
- Using the Statistical Facilities 11

PROGRAMMERS NOTES 15
- Introduction 15
- Overview 15
  - The Files 15
  - An Important Global Variable 16
  - Making Changes 16
  - Relinking 16
  - Transportability 16
  - Secret Codes 17
- The File G/.IVAN 17
  - Constants in G/.IVAN 18
  - The Program Run 18
  - The Procedures in G/.IVAN 19
    - Player Orientation Procedures 20
    - Other Procedures in G/.IVAN 23
- The File GameUn3 25
  - Constants 25
  - Procedures in GameUn3 25
- The File ItemFiler3 30
  - Important Constants 30
  - Important Global Variables 30
  - Procedures in ItemFiler3 32

PROGRAM MAINTENANCE 34
- The Disks 34
- Organization 34
- Disk Handling 35
  - The Game Code Disk--Before 35
  - The Game Code Disk--After 35
  - The Images Disk--Before and After 35

REFERENCES 37

APPENDIX A: Program Listing A-0

INTRODUCTION

Background and Problem:

Many student assessment procedures which are currently used in Navy training are not adequately accurate or consistent. This sometimes results in overtraining which increases costs needlessly, or undertraining which culminates in unqualified graduates being sent to the fleets.

Typical procedures for assessing performance do not adequately measure with sufficient fidelity, validity, and reliability real-world operationally oriented job-sample tasks. Consequently, student evaluation at its best is somewhat suspect, and decisions based upon this kind of assessment may be erroneous.

Better testing techniques are needed for assessing Navy trainees against performance standards employing tasks functionally similar to those encountered in operational contexts. One attempt to fulfill this requirement involves the use of microcomputer technology which is rapidly appearing in a number of Navy training and testing environments.

There is, however, no suitable knowledge base which can be tapped by the Navy (or others) for developing, evaluating, selecting, and using computer-based testing strategies incorporating graphic representations of job-sample tasks.

Many of these customary methods for measuring performance either on the job or in the classroom involve instruments which are primarily paper-and-pencil in nature, e.g., check lists, rating scales, critical incidences; and multiple-choice, completion, true-false, and matching formats.

A number of deficiencies exist with these traditional testing techniques, e.g.: (a) biased items are generated by different individuals, (b) item writing procedures are usually obscure, (c) there is a lack of objective standards for producing tests, (d) item content is not typically sampled in a systematic manner, and (e) there is usually a poor relationship between what is taught and test content.

What is required is a theoretically and empirically grounded technology of producing procedures for testing which will correct these faults. Very few data are presently available regarding the psychometric properties of testing strategies using microcomputer-based graphically represented simulations, models, or metaphors. Technical information is needed concerning the accuracy, consistency, sensitivity, and fidelity of these computer-based assessment schemes compared to more traditional testing techniques.

The objective of this exploratory development is to develop and evaluate microcomputer-based graphic representations of operationally oriented tasks to determine if they result in better assessment of student performance than more customary measurement methods. As a test-bed, microcomputer-based graphic models have been programmed to assess how well F-14 Pilots and Radar Intercept Officers (RIOs) recognize front-line Soviet and non-Soviet fighters and bombers.

A computer game based upon a sequential recognition paradigm has been designed and developed. It randomly selects and presents on the display of a microcomputer with millisecond speed either the front, side, or top views of four Russian bombers and ten of their advanced fighters. Also, the game management system can choose and flash corresponding silhouettes of NATO aircraft which act as distractors for their Soviet counterparts because of the high degree of similarity between them which could easily confuse U.S. aircrews.

This game, which is called FLASH IVAN (The F-14 community refers to the Russians generically as "Ivan"), assesses student performance by measuring: their "hit rate" or number of correct recognitions out of a total of forty-two silhouettes (half Soviet and the other half non-Soviet), the time it takes a student or latency to make a recognition judgment for each target or distractor aircraft, and the degree of confidence the student has in each of his recognition decisions. At the end of the game feedback is given to the student in terms of his hit rate, average response latency, average degree of confidence in his recognition judgments, and how his performance compares to other students who have played the game.

A file is maintained and available to the instructors which provides, in addition to these parameters for each student, recognition performance across aircraft for all students who played the game. This provides diagnostic assessments to instructors who can use this summative feedback to focus student attention on learning the salient distinctive features of certain aircraft in order to improve their recognition performance.

The game management system is programmed in a modular manner: instructing the student on how to play the game, retrieving and displaying individual images, keeping track of how well students play the game and providing them feedback, and linking these components by supervising routines in order to execute the game. This modularity in programming, together with the game management system's independence of any graphic database (e.g., ship silhouettes, human anatomy, electronic circuits, topography), contributes to its generalizability and transportability. The game, then, provides a set of software tools which can be used by others who need to assess recognition performance.

The software for the complete gaming system is currently on three floppy disks which control the play of the game, contain the graphic-images database, and maintain records of individuals' recognition performances. The game itself is run with only two dual-density disks on the Terak microcomputer employing two drives. It is implemented on the UCSD P-System and written in UCSD PASCAL. The disk placed in the bottom drive holds the actual game code; the disk placed in the top drive contains the independent graphic-images database. As soon as the system is booted, control is immediately passed to the game. Consequently, naive users need not deal with the nuances of the UCSD P-System. Recognition-performance data are saved for a number of individual players on the lower disk drive. A third disk containing game management facilities can be used by test administrators or researchers to format the recognition data to facilitate statistical analyses. Also, this third disk can be used to design a new game with a completely different set of graphic images to act as stimuli for recognition testing.

Empirical and psychometric studies will be conducted to ascertain if this computer-based game provides better estimation of student recognition performance compared to more customary measurement methods, i.e., multiple-choice or completion formats. These distinct assessment strategies will be evaluated in terms of their relative reliability, validity, and fidelity.

Objective

The objective of this technical report is to document the programming effort expended to develop and evaluate this generalizable and transferable computer-based gaming system for assessing recognition performance. This was done so that others who may want to use this set of software tools for either research, development, or operational implementation will have an easier time comprehending the modularity of the programming structure as well as how specific procedures can be adapted to suit a user's unique situation.

Inorder to create a context to facilitate further the understanding of the documentation of this computer-based game, the on-line instructions, presented to student pilots and RIOs whose performance will be assessed, are as follows:

"For research purposes, a computer game called "FLASH IVAN" has been designed and developed to assess how well Navy Pilots and RIOs recognize front-line Soviet and non-Soviet fighters and bombers. This randomly selects and presents on the Terak screen with millisecond speed either the front, side, or top views of four Russian bombers and ten of their advanced fighters. Also, the game management system can choose and flash corresponding silhouettes of NATO aircraft which act as distractors for the Soviet aircraft because of the high degree of similarity between them which could easily confuse U.S. aircrews.

"This game assesses student performance by measuring:

(1) your "hit rate" or percentage of correct recognitions out of a total of eighty-four silhouettes (half Soviet and the other half non-Soviet),

(2) the time it takes you or "latency" to make a recognition judgment for each target or distractor aircraft, and

(3) your degree of confidence in each recognition decision.

"At the end of each trial, you will be given feedback in terms of: the correctness of your response; a running tally of the number of correct recognitions, your hit rate, average response latency, and average degree of confidence in recognition judgments up to this point. At the end of the game, you will be given how your performance compares to other students who have played.

"Next, six examples will be presented to familiarize you with how the game is played. Notice that a silhouette will flash on the screen. If you do not pay attention and concentrate on the center of the screen you will likely miss seeing it! Your task is to identify as quickly as you can the flashed aircraft. After the image disappears, you will see the prompt: "AIRCRAFT NAME:". Use the key board to type in after this prompt what you think the aircraft is, i.e., its NATO name or corresponding alphanumeric designation, e.g., SABER or F-86. Misspellings count as wrong responses.

"Remember not only the correctness or accuracy but also the speed or quickness of your decision is crucial. The count-down clock for measuring the time it takes to make your recognition decision starts when you first see a silhouette. This clock is actually stopped when you begin keying in your response on the key board. Consequently, it is to your advantage to keep your hands hovering close to the key board to help minimize response time or latency.

"After you key in what you think the aircraft is, you will see a confidence or sureness scale. Use this to indicate the degree of confidence in your recognition decision concerning the silhouette you have just seen. Note that this confidence rating scale goes from LEAST CONFIDENT or 0% CONFIDENCE in your recognition decision on the left, to MOST CONFIDENT or 100% CONFIDENCE on the right.

"Following the prompt: "% RECOGNITION CONFIDENCE:", key in the cross referenced number which appears directly under the percentage of confidence which best reflects or approximates the amount of sureness you have in your judgment. For example, if you are absolutely sure that the silhouette was a FOKKER TRIPLANE you should key 0 which corresponds to 100% CONFIDENT; if you have absolutely no confidence in your recognition decision you should key in a TAB which corresponds to 0% CONFIDENT; and if you are somewhat sure in your judgment you may want to key in a 4 which corresponds to 40% CONFIDENT or 8 which corresponds to 80% CONFIDENT.

"After this you will be given feedback regarding the correctness of your recognition. If your response was incorrect, you will be given the proper identification of the aircraft which was just displayed in terms of alphanumeric designator and NATO name. Also, for all the silhouettes you have seen you will be presented with "% CORRECT RECOGNITIONS, AVERAGE RECOGNITION CONFIDENCE, and AVERAGE RESPONSE TIME". At the end of the game not only will these statistics be reported to you, but also your final score. This is a function of the above three variables. It is to your advantage to attempt to maximize the percentage of correct recognitions and recognition confidence while minimizing average response time. The system will then prompt you with "Hit RETURN and Watch the Center of the Screen for the Next Silhouette."

UTILITY FUNCTIONS--Creating New Games

1. The "ADM" Disk

This section of the documentation deals primarily with several utility functions that provide users the facility to manipulate the computer-based recognition gaming system to their liking. Included in this material are explanations of how to create new games with different sets of subject-matter images, and how to extract statistical data from the records of players' game performances. These facilities are all provided on a third disk labeled the "ADM" disk, also referred to

as the ADMINISTRATION Disk.

Instructors, on the one hand, may want to create a new game with a completely different database than the original game, FLASH IVAN, which uses aircraft-silhouettes. There are two basic steps in undertaking this task:

1) the creation of the computer images, and

2) the corresponding database which associates labels with each image.

The user-friendly programs SEMIPAINT and MAKEDIR on the ADM disk were designed to aid an instructor in performing this task.

Researchers or evaluators, on the other hand, may wish to extract statistical data from the game. The program MAKESTATS has been designed for this purpose.

For the programmer who is enhancing the game to suit an instructor's needs, the ADM disk provides many basic utilities, e.g., PRINT programs and disk formating programs, as aids. The actual game code, written in UCSD Pascal, also resides on the ADM disk. A programmer may want to change the code in making basic game changes. Of course, a basic familiarity with the UCSD P-system and UCSD Pascal (Bowles, 1977; Grogono, 1980; SoftTech, 1978) is prerequisite to successful completion of such a task.

The following sections give the details of the utility functions on the ADM disk. We have decided to approach the matter from the point of view of the user. Rather than describe each utility function separately, we have opted to group descriptions of the utilities together in the context of the two most important outcomes of their usage; hence, the two section headings:

1) Creating a Game

2) Using the Statistics Package

We recommend that you walk through the running program while simultaneously reading these sections.

2. Creating a Game

Two of the variable components that are the basis for a new game are :

1) the images, graphic representations, or pictures

2) the information associated with each image

This game-specific information is always contained on a disk which is separate from the game-code disk, and is to be placed in the top-disk drive during run time. Theoretically then, a new game can be played simply by putting a new disk

in the upper disk drive and rebooting. This new disk would contain a new set of graphics (called FOTOFILES) and new corresponding information (called an IMAGE DIRECTORY).

2.1. Creating Game Images

The task of creating graphics or images is certainly the bulk of the work load in creating a new game; it involves the meticulous recreation of drawings or their like into computer images. There are typically two ways of undertaking this task : by use of a digitizer or by hand using some sort of graphics editor.

Certainly the fastest and most convenient of these two methods is by use of a digitizer, a type of camera which has the capability to project any image that it can "see" onto the computer screen. However, we have yet to discover a digitizing system that is compatible with the TERAK microcomputer.

We resorted to the slower of method of converting each image by hand (see SEMIPAINT instructions for more details); however, any graphics editor that can work on the TERAK and create 320 x 240 pixel images should work fine. It is also important that the name of the file holding the image end in the suffix ".FOTO". Any file ending in ".FOTO" is called by convention a *fotofile* and generally corresponds to a 320 x 240 packed array of boolean. Each member in the array is stored in memory as a bit and corresponds to one pixel (or dot on the screen). We also recommend the use of a grid-system in converting technical drawings to computer image so as to maintain accuracy. By placing a piece of see-through graph paper over the drawing and a cooresponding grid on the computer display, one can accurately translate the originai figure to the screen.

In order to maximize the number of images that can be used in a game, we have given the game creator the option of dividing each 320 x 240 pixel *fotofile* into thirds. These thirds are referred to as the TOPTHIRD, MIDTHIRD, and BOTTHIRD and consist of 320 x 80 pixels. Thus, either 1, 2, or 3 images can be stored on one FOTOFILE; when an image is flashed to the screen it will automatically be centered. Several restrictions pertaining to gaming images to be wary of are:

1) a maximum of 89 images are allowed in the game

2) a maximum of 50 FOTOFILES are allowed on the upper disk

These restrictions have been imposed due to the limited storage capabilities of the TERAK microcomputers.

When all of the FOTOFILES for use in the game have been created, it is necessary to store them on one disk which must be FORMATed and ZEROed beforehand (see instructions for UCSD P-system). The disk must also hold, in addition to the game FOTOFILES, the following standard files that the prototype game needs to access:

EX1.FOTO
EX2.FOTO
EX3.FOTO
EX4.FOTO
EX5.FOTO
FLAGS.FOTO
EAGLE1.FOTO
EAGLE2.FOTO
IN1.FOTO
IN2.FOTO
INSTRUCT.TEXT -- game instructions
NONE.FOTO

You must copy these files onto your disk. Two other files that you need not worry about, which will appear on your disk later are:

NEWNAMES -- the Image Directory
HISCORE.DATA -- keeps record of top ten players

2.2. Creating the Image Directory

Once all of the game images have been put on the special disk, it becomes necessary to create an Image Directory. The Image Directory is a list of 100 records that provide the main program with information concerning each visual stimulus in the game. Records 90-100 have been specifically reserved for system images. Records 1-89 are for your use, giving the main program information concerning game images. Each record contains:

a) two identification names associated with each image
b) the name of the *fotofile* which holds the image
c) where on the *fotofile* the image is stored

The program MAKEDIR has been especially designed in aiding the game creator in making an Image Directory. Before executing MAKEDIR, be sure the disk with the game *fotofiles* is in the upper disk drive. This action is necessary because the Image Directory also stores device-dependent information concerning where each FOTOFILE is on the disk, thus enabling the use of better and faster routines in projecting an image to the screen. In addition to this, the Image Directory is stored on the images disk.

Upon executing MAKEDIR, the user will see the following menu:

MAKEGAME OPTIONS:
1) EDIT DIRECTORY
2) CONVERT DIRECTORY
3) QUICKLIST
4) MAKE HISCORES FILE
5) QUIT

Typing "1" will enable the user to begin creating an Image Directory for the first time or to edit a pre-existing Image Directory. The next prompt a user will encounter is:

Edit OLD file or make NEW file? [O/N] —>

Type "N" (meaning "NEW") to get the next prompt:

EDITING OPTIONS:
1) INDEX CHOICE
2) AUTO-INDEX

The INDEX CHOICE option allows the user to edit any one record in the index range 1 to 100. This option becomes especially useful to a user who is making small changes to an OLD Image Directory. The AUTO_INDEX option, on the other hand, will automatically loop through a predefined sequence of records after the user is done editing a particular record. This option is especially applicable to the user who is creating a NEW directory. After typing "2" specifying the AUTO-INDEX option, the next prompt to appear will be:

Enter lower index bound, space, upper index bound

These bounds indicate the range of records you wish to edit. Needless to say, the lower index bound should be less than the upper index bound, and both bounds should be within the 1-100 range. More likely than not, you will not need to edit records 90-100; they have been preset and pertain to "system" images. Be wary that once you begin editing records in a certain range, you must complete the sequence if you wish the information to be recorded on the upper disk. Eighty-nine records are a lot of records to edit in one sitting. If you have a limited amount of time you may only want to edit records in sequences of 10. This method will also allow you the freedom to go back and repair minor mistakes you may have made with the INDEX CHOICE option (as opposed to having to go through all 89 records before coming back to repair mistakes). Once you have entered your index bounds, you will be presented with a menu which cooresponds to one record in the Image Directory. The menu will appear as such:

INDEX NUMBER	*n*
Name 1:	none123
Name 2:	none123

Fotofilename:	FLAGS.FOTO
Fullscreen[T/F]	TRUE
TopThird [T/F]	FALSE
MidThird [T/F]	FALSE
BotThird [T/F]	FALSE

Use the arrow keys on the right side of the keyboard to move among the choices. You will notice a small arrow on the left border of the menu specifying which item you are currently pointing to. Type "S" to select the item you wish to make changes to. For example, suppose the indicator arrow is pointing at "Name 1:". Typing "S" will provoke a new prompt occurring at the bottom of the screen:

Name 1 is currently "none123"
Enter the new Name 1: -->

After entering the new Name 1 followed by <RET>, you will notice the new prompt on the right hand side of the screen:

Change more values? [Y/N]

This same prompt will occur after any change that you make. A "Y" response will bring you back to the same indexed record. A "N" response will automatically project the next sequential record to the screen (provided you are in the AUTO-INDEX mode). So when typing "N" be sure that you have entered in all of the correct information, because if you have make any mistakes and typed "N" going on to the next record, you won't be able to go back and correct the mistakes until you are done with the sequence of records. Suppose you were to type "Y", going back to the same record to edit. You select "Name 2:"; if there is no second name associated with your image, it is best to enter an empty string by simply hitting <RET> when prompted for "Name 2:"; otherwise, if a game player were to respond incorrectly to this game image, the name "none123" would appear under "the correct name is:" heading.

In selecting "Fotofile name: ", you will notice that it has been preset with the name FLAGS.FOTO. This acts as a default file which will be flashed to the screen if you happened to have forgotten to type in a fotofile name. When you enter the fotofile name, be sure to include the ".FOTO" suffix. All fotofiles are assumed to be in the top drive so the prefix "#5:" is unnecessary.

The remaining four fields of the record indicate where the image is stored on the fotofile. For instance, if "Fullscreen" were set to "TRUE", then image is contained on a complete fotofile; and conversely, it would be set "FALSE" if it was not contained on a complete fotofile. Note that more the one of these fields could not possibly be set "true" at the same time; in other words, an image could not possibly occupy a full *fotofile* and a third of a *fotofile* simultaneously. So as soon as as one of these items is selected, it is automatically set to "TRUE" while the remaining items are set to "FALSE".

Whenever you are done editing after either completing a AUTO INDEX sequence or responding "N" to the prompt "Another?" in INDEX-CHOICE, the screen will clear and the computer will inform you that it is "converting" the directory, before taking you back to the main menu. During "converting" if any of the *fotofiles* that you listed under "Fotofile name:" are not on the images disk, you will get a message notifying you of this. This function is described in more detail below.

2.3. More Utilities

After Editing an Image Directory, or whenever running the utilities program MAKEDIR, the user will always be presented with the main prompt:

MAKEGAME OPTIONS:
1) EDIT DIRECTORY
2) CONVERT DIRECTORY
3) QUICKLIST
4) MAKE HISCORES FILE
5) QUIT

We have already discussed option #1 concerning Editing a directory. The following paragraphs discuss the remaining options.

2.3.1. Converting the Directory

Typing "2" from the main prompt line will run code that "converts" an Image Directory. Converting a directory is processed on the top disk which translates *fotofile* names into numbers which describe where a *fotofile* is on disk. This number (referred to as the BLOCK number) is stored in a "hidden" field in each record of the Image Directory; its main function is to speed up the time it takes to access an image from disk and flash it to the screen during game time. When it has finished, image directories ("NEWNAMES") will be written to both the upper and lower disks.

Whenever any disk operations (ADDing a file, DELETING a file, KRUNCHing the disk, etc.) are performed on the images disk, the P-system filer usually rearranges the placement of files on a disk; thus Converting the disk is essential in these instances so as to assign new BLOCK numbers to Fotofiles. If ever you come across a bizarre collage of images flashed to the screen during game time, it has probably resulted from your forgetting to "Convert" the Image Directory.

Note that it is unnecessary to select the CONVERT option if you are editing an Image Directory since the EDITING option automatically converts the disk for you. Also, if ever you entered a *fotofile* name that is not on the images disk,

the Conversion function will send a message to the screen indicating this.

2.3.2. Quicklist

Typing "3" from the main menu will automate the QUICKLIST function. This is a convenient way to quickly look over every record in the Image Directory. As each record is scrolled down the screen, you will notice the addition of the aforementioned "hidden" field labelled BLOCK included in each record listing. The entire list of 100 records in the Image Directory will be sent to the screen, and then to a file "QUICKLIST.TEXT" on the disk in the bottom disk drive. This list can then be sent to the line printer for further scrutiny using the PRINT program.

2.3.3. Making the Hiscores File

"Make Hiscores File" will create a new HISCORE.DATA file which will prompt you for the top ten players (we have used fictional people) and their respective scores. When entering in the new list, it is not necessary to list players and scores in any special order. The program will automatically list them from top to bottom in descending order according to score. The current version of the game maps scores in the 0 - 1000 range, so it is best to enter scores in this range.

3. Using the Statistical Facilities

MAKESTATS is a program which takes the data from the computer recognition game and formats it into a text file so that it can be viewed or sent to a printer. In order for it to operate correctly it must have two important files on the same disk (i.e. the "ADM" disk):

NEWNAMES
GAMES.DATA

NEWNAMES, the image directory, is needed so as to associate image names with statistics. *GAMES.DATA*, a record file, holds the game stats and consists of the record type "gamestats":

```
const  numberpictures = 89;
type   GAMESTATS  = record
         name     : nametype;
         SS       : sstype;
         date     : nametype;
         latency  : array[1..numberpictures] of integer;
         confidence : array[1..numberpictures] of scale;
         correct  : array[1..numberpictures] of boolean;
       end;
type   nametype   = string[15];
sstype           = string[11];
```

The constant NUMBERPICTURES is set to the number of total possible images which can be shown in the game, not the actual number used in each game. Makestats is set up so that any number of graphic stimuli can be used (up to the maximum) in a game. Only those actual photos used in each game are tallied for averages over several games. Records of type GAMESTATS keep statistics for each game and when it is through, and are saved in a disk file, GAMES.DATA. Recorded for each game are:

name:	The player's name, up to 15 letters;
ss:	The player's Social Security number, any string up to 11 characters is allowed so that errors can be avoided when non-numeric data is entered;
date:	And likewise, a 15-character length string for the date the game is played is kept;
latency:	The player's response latency is kept for every image he responds to, if an image is not used in the game then the latency will be 0;
confidence:	And similarly, the player's confidence rating which he has keyed in for every image;
correct:	Whether the player actually got the image recognition correct or not.

After a game is completed, a variable CURRENTGAME of type GAMESTATS is appended onto GAMES.DATA.

When Makestats is executed, GAMES.DATA is opened and each game that has been saved is read one at a time. They are then neatly formatted and put into two textfiles on disk: LATENCY.TEXT and CONFIDENCE.TEXT. Latency.text will include vertical and horizontal averages of the response latencies. That is, each player's average response latency over one game, and the average response latency for each photo over all the games recorded. Confidence.text includes the same thing for the confidence ratings and averages,

but the correctness ratings are also included; a "+" is put before the confidence rating if the player got the recognition correct, otherwise a "-" if he got it wrong. Also, in the formatted output are included the percentage of graphic stimuli the player correctly recognized in each game and the percentage of games which got any certain photo right.

As makestats is executed, the old data in GAMES.DATA is erased, and a new GAMES.DATA is initialized with 0 game entries. In order for the game to save statistics properly, GAMES.DATA *MUST* be the *LAST* file on the disk's directory. Otherwise, the file will get too large, there will be an I/O error, and the game will not be saved. If the error message appears:

i/o error: no room on volume

Then you must:

1) Delete GAMES.DATA from the disk,
2) K)runch the disk in the Filer,
3) eXecute DRIVER.

DRIVER is an executable file which initializes an empty GAMES.DATA file on the disk in the upper disk drive.

The text files LATENCY.TEXT and CONFIDENCE.TEXT have cross-references to index planes and players. This is a sample LATENCY.TEXT printout:

LATENCIES (in milliseconds)												
PLAYER/#	PLANES											
	1	2	3	4	5	6	7	8	9	10	11	12
	-	-	-	-	-	-	-	-	-	-	-	-
AA	2612	948	902	1388	1948	801	86	382	8001	2845	288	1002
AB	8482	2884	90	456	1884	1288	288	948	48	288	8828	8228
	13	14	15	16	17	18	19	20	21	22	23	24
	-	-	-	-	-	-	-	-	-	-	-	-
AA	884	824	288	1388	388	128	906	4858	488	184	1884	444
AB	48	2884	2888	482	284	2844	844	28	284	2844	1448	1200

	NAME	SS #	DATE	Average Latency
AA:	Lee, J D	000-00-8488	12-8-1884	1088
AB:	Applebee 'ee k	100 28 1288	8 5 85	2808

	PLANES:	Average Latency
1. (Top)	FRESCO	2884
2. (Top)	FARMER	484
3. (Top)	FITTER	802
4. (Top)	FLAGON	1882
5. (Top)	FISHBED	488
6. (Top)	FISHPOT	8488
7. (Top)	BLINDER	482
8. (Top)	BADGER	188
9. (Side)	FRESCO	882
10. (Side)	FARMER	8822
11. (Side)	FITTER	284
12. (Side)	FLAGON	888
13. (Side)	FISHBED	801
14. (Side)	FISHPOT	811
15. (Side)	BLINDER	122
16. (Side)	BADGER	2884
17. (Front)	FRESCO	484
18. (Front)	FARMER	854
19. (Front)	FITTER	801
20. (Front)	FLAGON	848
21. (Front)	FISHBED	858
22. (Front)	FISHPOT	882
23. (Front)	BLINDER	882
24. (Front)	BADGER	1882

The players are indexed by capital letters AA,AB,AC,...,BA,BB...ZZ and later are cross referenced with their name and Social Security number, the date of the game, and their average response latency (or average confidence and percentage correct as in CONFIDENCE.TEXT). The images are indexed with numbers 1,2,3... across in rows of 12 so that it is feasible to put all the data on one page. At the end the planes are cross-referenced to the viewing angle (top, front, or side), the plane name, and the average response latency scored on that plane for ALL the games using that graphic image. The same pertains to the average confidence and percentage correct for ALL the games using that picture, for CONFIDENCE.TEXT.

PROGRAMMER'S NOTES --Modifying the Game Code

4. Introduction

This section of the documentation is designed to serve as a technical reference for programmers who may wish to make slight modifications to the game code itself which is independent of the database. We have divided it into four main sections:

1. Overview	--an introduction and reference guide to making changes
2. The File G/.IVAN	--technical descriptions of procedures
3. The File GAMEUN3	--technical descriptions of procedures
4. The File ITEMFILER3	--technical descriptions of procedures

5. Overview

The computer-based recognition game in its present form, *Flash Ivan*, is currently implemented on the UCSD p-system, version II.0. It is run on a Terak 8510 dual-density machine, with an auxiliary 8515 dual-density drive. The game requires two disks, the bottom (#4:) drive gets the disk with the actual program (named *System.startup*-- a program that automatically runs when the disk is inserted), while the top (#5:) drive gets the disk with the database, described in the section on *ItemFiler3*.

This particular implementation of the game consists of aircraft silhouettes, but it could be used for any set of graphic images that could be drawn into the database (see the *Utilities* documentation), and used in a recognition-game format. Because of this, we may refer to "planes", "airplanes", "pictures", "images", "visual stimuli", "graphics", or "database objects". All of these refer to the same thing.

5.1. The Files

The Flash Ivan game code consists of the following files:

G/.IVAN	--	Pascal host program
GAMEUN3	--	Pascal library file
ITEMFILER3	--	Pascal library file
ERROR	--	Assembler code for sounds
CLICK2	--	Assembler code for sounds
TIMEPI	--	Assembler code for sounds

The code was split into seperate files to make manageable segments, without much effort being made toward extreme modular cohesiveness. The host file, G/.IVAN, contains the main driving routine, and several assorted procedures and functions. The file GAMEUN3 contains more assorted procedures and functions, as well as a few constant declarations for good measure. The most cohesive file, ITEMFILER3, contains procedures dealing with the database, some constant declarations, and the HI-SCORES procedure. There are also some separate assembly language routines that need to be linked: ERROR, TIMEPI, and

CLICK2; these all produce the different sound effects.

5.2. An Important Global Variable

The variable **Info_List**, declared in the file *ItemFiler3*, is an array with one element for each item in the database.* Each element in the array contains information on what the airplane's different names are, and how to get it from the disk and show it on the screen. This is, in some sense, the "master variable" of the game. It allows most of the game procedures to think of the planes only in terms of indices into the array, and allow a couple of interfaces (the procedure *CheckAnswer*, the functions in the file *Itemfiler3*) to actually deal with the other information.

5.3. Making Changes

For programmers who plan on making any changes to these files, we recommend using the ensuing FLASH IVAN technical descriptions as a reference. Before relinking any freshly compiled files (listed above), please be sure the compilation dates as listed from the Filer are consistent. If the dates are not the same on any two files, linking errors will result. To change the date on any file, you must first change the date for the disk (Date option in the Filer) and then recompile or reassemble the file.

5.4. Relinking

When linking, remember that "G/.IVAN" should be typed in as response to the prompt "Host?". For the sequence of prompts "Lib file?", the other files listed above (as well as "*" for the System.library) should be included. For the prompt "Output file?", be sure to add the ".CODE" suffix so that it will be executable. On the official game disks, we often moved our executable game file into the file *System.startup*; thus upon booting, control is immediately passed to the game. In this way, naive game players need not deal with the particularities of the P-System.

5.5. Transportablity

Flash Ivan is designed to run on any computer with at least 128K RAM that supports the UCSD P-system. However, if Flash Ivan is to be run on any machine other than the Terak, slight modifications must be made to the game code. As a rule of thumb, it is safe to assume that any code having to do with device-dependent graphics or sound manipulations will have to rewritten. To make the game code as transportable as possible we have attempted to localize most of the machine dependent code in the file ITEMFILER3. Modifications must be made here. ITEMFILER3 serves as a home for the majority of the graphics code.

---

* We consider the *database* to contain all the information about the pictures, plus the pictures themselves. The top disk (#5:) contains the database.

5.6. Secret Codes

After having made any changes to some of the Pascal files, we suggest that you record the latest date of change in the string constants at the top of each file, *DateMain*, *DateGameUn3*, or *DateItemFiler3*. These three dates can be displayed from the linked and running version of the game by typing in a secret code word at the start of any game (see the function *Practice*. in the file *G/.IVAN*. With so many different files and so many different disks, as well as several programmers, we found that this facility helped us organize ourselves as well as see what version of the game we were actually playing.

In addition to seeing what version of the game you have, there are other functions you can invoke from the start of the game. Upon seeing the prompt "HIT RETURN TO BEGIN GAME" just above the Eagle's head, you can access the "version" function as well as a few other helpful ones. The code characters and their corresponding functions are listed below:

<esc>	--	to bypass instructions and examples
"v"	--	to list versions of Pascal host and objects
"m"	--	to see memory available. We were pushing the upper limits of RAM when this documentation was written, so this function came in handy. Be wary if you plan to make any major additions.
"d"	--	to display any pictures from the image directory The image directory is an array of 100 records containing graphical information on each game image. Entering numbers between 1-100 is advised here. If any images are centered incorrectly or the wrong picture is displayed, it is likely that the image directory needs to be "converted", or bad information was put in the image directory by a game maker. Consult the "Flash Ivan Utilities" Documentation.
"h"	--	to view the HISCORES file

These characters can entered in either upper or lower case. One version of the game requires you to hit the password "boatman" from the main prompt in order to access any of these functions. This prompt *must* be lower case. Since the Terak is initialized to an "all-cap" status, you need to know how to get to an "upper/lower case" status. The <DC2> key at the lower right of the keyboard provides the function of toggling between these two keyboard states. See the procedure *Practice* in the file *G/.IVAN* for more details.

6. The File G/.IVAN

G/.ivan is a file that contains the main body of the game program. It makes calls to other procedures defined in library files so as to provide a cohesive unit among all of the game files.

6.1. Constants in G/.IVAN

There are three constants defined in this file. It is arguable whether or not the constant declarations should be here instead of in one of the units with the other constants, but they are only used here, so there is some justification.

name: This is set to the filename where the statistics will be collected.

PRACSTART: This is set to the beginning index of the practice pictures. In our implementation, the actual game pictures go from 1 to 84. Practice pictures then start at 86. 85 is a "delimiting" entry. (See the section on *ItemFiler3* for an explanation of the data base, indexing, etc.)

numberpictures : This tells the statistics functions how many different test items there are. Statistics will be printed for items *one* through *numberpictures*. (See the *MakeStats* subsection of *Utilities* for more on the statistics functions.)

DateMain: This is for programming convenience. It is a string constant set to the date and time the file *G/.Ivan* is modified. The "v option at the beginning of the game will print out this constant, as well as similar ones in *GameUn3*, and *Itemfiler3*.

6.2. The Program Run

The "main" procedure of the program is fairly small (about ten lines), and is run through only once per game. Calls are made to procedures to initialize the statistics variables and database list (*Info_List*), and show the opening animation. Then the variable *TotalPictures* is set to forty-two. This is a number particular to this game, and means that only forty-two of the total of eighty-four pictures will be included in any one game. For a game to include the entire set of pictures, a call to the function *ListLength* with the parameter *Info_List* to set *TotalPictures* could be made instead.

Next, the procedure *ChoosePlanes* is called, with *PicSequence* as its parameter. *PicSequence* is an array of integers, declared in GameUn3. The integers it will contain correspond to database indices, one for each picture contained there. The *ChoosePlane* procedure is another that is specific to the database, and particular game demands of **FlashIvan**. It will pick seven Soviet top views, the seven corresponding NATO distractor top views; seven Soviet sides, the corresponding NATO side view distractors; and seven Soviet front views and their distractors for a total of forty-two aircraft images. This is out of a possible eighty-four silhouettes. They will not be randomly *ordered*, but each set of seven will be randomly *chosen* from fourteen possible images.

In order to present the pictures in a random order, the procedure *Shuffle* is next called, with reference parameter *PicSequence*, and value parameter *TotalPictures* to tell how many to shuffle. *PicSequence* will return with the same set of picture indices, but in a new, shuffled order.

The procedure *PrivacyAct* shows two fotofiles which contain the necessary text explaining to the research subjects, who are about to play the game, that they are asked to not only identify themselves but also give their social security numbers to facilitate statistical analyses involved in evaluating this computer-based testing strategy. Further, the subjects are informed that playing the game is completely voluntary on their part. This procedure will also present a textfile, one screenful at a time, containing instructions for the game, and any other preliminary comments that the game player should be familiar with. Someone implementing their own game could write their own version of the instructions. The file should be on the top disk (#5:), and be called **INSTRUCT.TEXT**.

Hello will prompt for and read the player's social security number, name, and date. It will then re-display the information and ask for confirmation. The player is allowed to re-enter information until he is satisfied with it.

The *Practice* procedure first shows three examples, animating or mimicking a game so the player can see how to play, and in what order, including the computer typing in, instead of the subject, the names of aircraft displayed character by character. Then, it calls the same procedures for showing three additional example silhouettes to elicit actual practice responses from the players, i.e., typing in themselves aircraft names, and for reporting feedback to them as the actual game would for three more example trials. This allows the subject to become more comfortable with how the game is played before she/he really attempts it. The six example trials consist of the same set of pictures every time. (See *Utilities* for an explanation of how to put in practice pictures.) This procedure does not save the results when done.

InitStats initializes the statistics variables. For a more detailed description, see below.

The procedure *Game2* is the major game-playing loop which presents the player with the full set of images (whatever the variable *TotalPictures*, which resides in the file GameUn3, says) and keep track of all the game information.

The procedure *AfterGame* will write the information for the game just played to the disk under the subject's name. It will then show the player's final score, and call *HiScores* to show the current top-ten players.

6.3. The procedures in G/.IVAN

Due to the space limitations of the Terak 8510 computers, we had to make as many procedures as possible "Segment" procedures. That is, they will only be loaded into memory when needed, and then moved back out leaving room for others.

6.3.1. Player Orientation Procedures

These are the procedures that are used in the beginning of the game to acquaint the user with the rules, and allow him/her to get familiar with the flow of the game by watching and playing some practice examples.

Procedure *PaintBlock(VAR Source, Srcwid, Srcx, Srcy, integer; VAR Dest; Dstwid, Dstx, Dsty, Cntx, Cnty, Mode, Gray: integer);*

This externally assembled file is located in the *System.Library*, and can thus be accessed upon linking to the *System.Library*. *PaintBlock* simply copies bit maps from "Source" (a boolean array) to "Dest" (another boolean array. "Cntx" and "Cnty" are the width and height of the block of the boolean array to be copied. The "Mode" parameter gives the following boolean operations: 0 = store, 1 = or, 2 = and, 3 = xor, 4 = complement. The "Gray" mode seems to work best set to -1.

Procedures *Click2, Timepi, Error;*

These procedures are external MACRO-11 assembly procedures used for various game sound effects. For assembly code alterations, refer to the UCSD Pascal User's Manual (SofTech), or your favorite MACRO-11 (PDP-11 Assembly Language) handbook. Essentially, all the sound effects are produced by switching bit 7 of the VCR (Video Control Register) at various frequencies. The include file *SND_EFF.TEXT* is inserted at the beginning of each of the three assembly procedures and contains two simple MACRO algorithms for switching the VCR.

If one is interested in creating or adapting some new sounds, methodical experimentation with with these macros is suggested.

IMPORTANT: When linking assembled procedures to Pascal host programs, make certain that all file dates (including the include files) are the same. If they are not dated similarly, the Linker reports a diagnostic such as

"Click2.code not found"

In addition, all assembled procedures must NOT be linked to the System.Library before linking to the Pascal host; the respective code file will also be reported "Not Found" by the Linker.

Procedure *Animate;*

This Pascal procedure's two main purposes are to initialize the boolean array used for graphics, and to perform the opening animation sequence at the beginning of each program run.

Animate first initializes the two global boolean arrays *Minifoto* and *Crosshairs* to contain their bit-map icons for the duration of the game. *Minifoto* contains the labels "AIRCRAFT:" and "% CONFIDENCE" as well as the Confidence ruler "TAB .... 0"; *CROSSHAIRS* contains the gun sight icon that is always flashed to the screen a split second before an actual game recognition

image (see procedure *Display*). Both arrays, *Minifoto* and *Crosshairs*, are initialized by a disk read from a FOTOFILE indexed "91" in *Info_list* to the *Screen* buffer, followed by two calls to *Paintblock* copying both bit-map arrays from *Screen*.

After these initializations, the Eagle animation code follows. The animation is the simple "flip-book" approach centered upon the Eagle's head; it is accomplished by a series of calls to *PaintBlock* with an interspersed call to the sound effects procedure *TimePi*. Notice that each call to *PaintBlock* is followed by a call to *UnitWrite(3, Screen, 63)* so that the screen is updated for each animation "frame".

Procedure *Instruct;*

Instruct is the first nested procedure in *Practice*. Its primary function is to read the contents of the file *Instruct.Text* (the game instructions) from the upper disk drive. You may notice that whenever game-specific information is read into the game (such as reading in the main game array *Info_List*, the game instructions, or the FotoFiles) they are always read in from the upper disk (#5:). We implemented this standard in our game to make it flexible to new games. *Instruct* reads in one string at a time from the text file, and then outputs each line to the screen one at time. This type of implementation prevented us from having to read in the whole textfile, thus saving valuable memory space needed for game code. After 21 lines have been projected to the CRT, no more lines are read from the file until the game player hits the <RET> key.

Procedure *Play3;*

Play3 simulates three game examples exactly as they would appear in the game. The variable *ListIndex* is set to 95, 96, and 97 in a loop. These numbers correspond to the practice game images' indices in the main game array *Info_List*. Records 90-100 in *Info_List* have been reserved for such purposes as storing information pertaining to these practice game images and other system images such as those used in the opening Eagle animation. The calls to *Display* and *SingleTrial* flash the image to screen, prompt the player for a response, collect statistics, and display the results. These 2 calls are the very calls used in the actual game as well. The additional variable *FakeList* is used in this procedure to hide the fact that list indices > 90 are being displayed. *Single_Trial* only expects to be called with numbers less than 90, the total amount that can be used in a game. Throughout the *Practice* module, *FakeList* is set between 1 and 6, so that statistics can tabulated for six examples. These "fake" statistics stored in the *scoresfile* are overwritten during the first 6 loops through *SingleTrial* during the actual game (see Procedure *Game2*).

Procedure *Answer(VAR A1: string; X, Conf: integer);*

This procedure is used in looping through the 3 automated examples in *Practice*; it attempts to simulate *SingleTrial* by displaying prompts, collecting statistics, and displaying results. In addition to this, *Answer* also simulates a player, by supplying the responses (*A1*) as well. *X* is the number of characters in the string A1; between each character being output to the CRT, *Click2* is called to simulate the sound of the keyboard. *Conf* is the confidence integer to be recorded in the statistics.

Procedure *Practice;* (main block)

This is the main procedure for providing game players practice with game examples; it makes calls to the aforementioned procedures which are nested in *Practice*: *Instruct, Play3,* and, *Answer.* Notice that this procedure can be immediately exited with the entry of a password from the standard input, when the prompt to "hit <ret>" comes up. This drops the program into a loop where a response of <esc> to the new prompt initiates the game, bypassing instructions and examples. We implemented this "secret" option, so as not to needlessly walk players who know the game sufficiently well through the instructions and examples. There are also four other options at this point. The user could type the character "H" (upper or lower case) to see the Hi-Scores board, "V" to see the versions of the three game files (the constants defined in each file-- *DateMain, DateGameUn3, DateItemFiler3*), "M" to see memory available (via calls to the provided function *MemAvail*), or "D" to display any pictures from the database. For this option, the user will be prompted for the index number of the picture to display. (Note: this option is not currently implemented.) This is all put into a loop, so whenever the prompt to "hit <ret>" comes up, the user can continually hit the character options instead. The loop drops through when either the return key or the escape key is hit.

Notice that *FakeList* has been initialized here before going through the 6 game examples. *FakeList* is used as a dummy index so as to keep statistics during the practice run. These statistics will be overwritten and forgotten during the real game run. After this initialization, the automated examples are then created with 3 calls to *Display* and *Answer.* Notice that indices 92, 93, and 94 are used here. These integers correspond to the game example information stored in *Info_List. Play3*, the participatory examples, is then called; it uses *Info_List* indices 95, 96, and 97.

Procedure *PrivacyAct;*

For the instructions, this simply opens the file "#5:instruct.text" and reads one line at a time, then writes that line to the screen. Every 21 lines, the "write a line" loop stops, writes "hit <RETURN>", and waits for the return key to be hit with a "readln()" statement.

The text of the privacy act has been typed in to two fotofiles. To show these, it does a UnitRead of the fotofile from the disk, and a UnitWrite to the screen.

Procedure *Hello(VAR player: nametype; VAR date: nametype; VAR SS: sstype);*

The procedure *Hello* prompts the player for his name and the date, both of which are of type *NameType*, a string of 15 characters in length; and also for his Social Security Number, of *sstype*, a string 11 characters long. All three are stored at the head of the game statistics file *GameStats*, right before the arrays which store response latency, correctness, and confidence. The *PrivacyAct* function is also called in *Hello* and the prompts to start playing the game are printed out.

6.3.2. Other Procedures in G/.IVAN

Procedure *Aftergame;*

This procedure saves the *GameStats* file on the end of the *Games.data* file on disk when the game is over. It also prints out the player's final score and calls *OutputStats*; and also calls *HighScore.*

Procedure *OutputStats;*

This procedure prints out for the user his/her percentage correct recognitions, average recognition confidence, and average response latency.

Procedure *Initstats*;

This procedure just initializes all of the arrays and variables used within the statistics portion of the program, invariably to 0.

Procedure *AfterPicture*;

This procedure puts the player's response latency, confidence, and correctness in the proper spot in the GAMESTATS statistics file, and keeps track of various variables, such as how many planes have been shown, which are used to output statistics to the player.

Procedure *GetConfidence(VAR conf: integer)*;

This procedure displays the confidence-rating continuum or scale and prompt via the calls to PaintBlock, and a couple of "write" statements. It then reads the player's response as a character. If the character is not either a TAB,

or a digit, then the user will be asked to try again. Once a valid character is entered, it is converted to an integer value, percentage of confidence in the recognition judgment. For TAB, the confidence is returned as 0%. A zero keyed in corresponds to 100%. All other digits are 100 times their value. This is reflected in the recognition-response confidence-rating continuum.

Procedure *SingleTrial(ListIndex: integer; ifprac: boolean)*;

The *SingleTrial* procedure times the player's response, checks the answer for correctness, and calls the appropriate statistics routines to keep track of the player's scoring. It is called from *Game2* just after a picture has been shown. The parameter *ListIndex* is the index into *InfoList* for the picture just shown. It needs this to be able to look-up information in *InfoList* about that particular picture. The other parameter, *ifprac*, tells the function whether or not to look at the practice set of pictures (starting at InfoList[PRACSTART]), instead of the "real" set of pictures. This is needed because the practice procedures at the beginning of the game also need to call *SingleTrial*, but with a completely different set of pictures.
The delay loop:

for i := 1 to SetSpeed *do* DELAY(250);

determines how long the picture will remain on the screen. *SetSpeed* is a constant, defined in *GameUn8*. It allows coarse control over the delay amount (the DELAY() procedure delays for approximately one one-hundredth of it's parameter: thus a change in *SetSpeed* of one results in a delay change of about 2.5 seconds). When the delay loop is through, the *page(output)* command will clear the picture. Then a prompt is shown, and the player's response is read into the string *guess*. *Ticks* will then contain the number of "machine ticks"* that occurred between the disappearance of the picture, and the typing of the second key by the game player. This approach was chosen to in an attempt to not penalize poor typists, yet still get some measure of the player's response time.

The calls to *AfterPicture* and *OutputStats* take care of updating the statistics, and showing the player his current performance information. After this is printed out, the player is given a chance to see the picture again, and look at it for as long as he wants.

The section of code at the end, currently commented out, will allow only only a ten second pause after the end of the current trial. If the player does not type the <return> key before ten seconds are up, the game will write a message to the screen telling the player to pay attention, then the game will continue. As it is now, the game is set-up to remain in a wait-state if the return key is not hit.

*(ticks/*60*) times *1000* equals the time in milliseconds.

Procedure *Game2*;

This procedure is called *Game2* for traditional reasons (once upon a time there was a *Game1*...). As mentioned earlier, this is the major game-playing loop in the program. It loops from 1 to *TotalPictures*, a variable set in the main program body. In our implementation, we set it to 42, so we always get a game of 42 trials. The loop counter is used to index into *PicSequence*, a previously loaded array of integers which are in turn passed one at a time to *SingleTrial* and used as indices into *InfoList*. These integers are unique, range from 1 to the highest possible game picture, and have been chosen and shuffled in *main*.

7. The File GameUn3

The file *GameUn3* serves as a home to many of the assorted functions needed for the program. It is not a cohesive module in the software engineering sense. Only *ItemFiler3* approaches that.

7.1. Constants

The same comments about the constants in *G/.Ivan* hold here, also.

DateGameUn3 : This is a string telling when the file was last updated. See the constant *DateMain* in the section *G/.Ivan*.

MaxInt : This represents the largest *positive* integer that the Terak can hold. This is why there is a limit to the reported latency of *MaxInt* milliseconds, or about thirty-two seconds.

SetSpeed : This is used to roughly effect the amount of time each picture is shown. A larger number will show the picture for a longer amount of time. It is used in *SingleTrial*.

ChooseGame, All_In_One, FlashGame : These are all booleans which are supposed to allow different game setups. At this time, none of this is implemented.

7.2. Procedures in GameUn3

Procedure *Randomize(VAR seed: integer);*

This is an external function, found in the supplied *System.Library*, which fills the integer *seed* with a number derived from the system clock.

Function *Random(VAR seed: integer; Low, High: integer): integer;*

This random function returns an integer between (and including) the two bounds *Low* and *High*, and changes *seed* as well. This function is derived from information given in the book "Fortran 77 - Principles of Programming" by Jerrold L. Wagener, in chapter 8. *Random* has a period of 1024 (meaning the sequence of numbers generated will not repeat until 1024 calls have been made), and is designed for a machine with 16 bit integers.

The procedure was tested for approximating random selections by choosing 7 items from a possible 14. The results were tabulated, and the selection was done repeatedly. This test was done 10,000 times. The results follow:

item number	how many times chosen
1	4748
2	5463
3	5422
4	5092
5	4712
6	5220
7	4963
8	4297
9	5758
10	4773
11	4793
12	4928
13	4841
14	4990

These findings indicated that the pseudorandom number generator did indeed approximate random selections. The expectation of each item number for 10,000 trials is 5,000 which was approached by how many times each item number was chosen by the generator.

Procedure *Shuffle(VAR IndexArray: IndexList; Num_of_Pics: integer);*

The input reference parameter *IndexArray* is a set of indices into *InfoList*, previously chosen, but not necessarily in a mixed order. *Shuffle* will randomly choose 200 pairs of indices into *IndexArray*, and then exchange their contents. After *Shuffle* is called, sequential accesses into IndexArray will yield a random sequence of the original set of numbers.

Procedure *MakeSequence(VAR IndexArray: IndexList; Num_of_Pics: integer);*

This procedure is not currently used in our set up, but is more general than the procedure we use to make a game sequence (*ChoosePlanes*). After a call to *MakeSequence()*, the parameter passed in as *IndexArray* will contain a random

sequence of integers from 1 to whatever was passed into the second parameter (*Num_of_Pics*), each integer appearing once. This is useful for games where one game consists of each and every picture showing up once and only once. *MakeSequence* calls *Shuffle* to actually do the mixing.

Procedure *ChoosePlanes(VAR IndexArray: IndexList; Num_of_Pics: integer)*;

This is a more complicated procedure for composing a set of silhouettes or pictures for an instance of a game. For **Flash-Ivan**, we had a total of 14 Soviet aircraft, each with a top, a side, and a front view. For each of the total 42 (14 times 3) Soviet pictures, we also had a corresponding NATO picture. We chose this picture to look as similar to the Soviet one as possible, to act as a "distractor".

We wanted this game to show 42 silhouettes in an unpredictable order. These 42 images should include equal numbers of fronts, sides, and tops, and equal numbers of Soviet and Non-Soviet aircraft. Further, for each Soviet silhouette shown, its matching distractor should also be shown sometime during the game.

ChoosePlanes relies on a special ordering of *Info_List* (corresponding to the ordering in the database). There should be 14 Soviet planes of one view (items 1 through 14), then the 14 distractors for those pictures in the same order (15 through 29), then 14 Soviet planes of another view, etc. This makes the relation between any picture and its distractor very simple. Just add 14.

ChoosePlanes has three sections. Each section chooses the fourteen pictures for one view. The sections are the same, except that different bounds are passed to *random* to reflect the new set of pictures to choose from, and each section fills a different piece of the array parameter *IndexArray*. Each section itself is a seven-iteration "for" loop. Each iteration chooses two pictures: a Soviet and a distractor. The Soviet picture is chosen by the random procedure in the specified bounds, then the distractor is found by adding fourteen. These two numbers are stored in *IndexArray* at consecutive locations.

One possible problem is that the random function could happen to return a number that it has already chosen. To take care of this, we declare a set of integers, *AlreadyChosen*, which is checked each time a new number is generated. If the new number is not in the set, then it is put into the set and the procedure goes on as described above. If the new number is in the set, then a loop is started. This loop generates a new number in the same bounds, and checks again. It continues until it finds a number not yet chosen. Although this method has the possibility (very slim) of continually choosing numbers already chosen forever, it was found that the time it actually took was never noticeable.

Procedure *UpperCase(VAR Name1: string)*;

This procedure checks each character of *Name1* and, if it is a lower-case alphabetic character ('a' through 'z') it converts it to its upper-case representation by subtracting decimal 32 from its ordinal value. This procedure works assuming an ASCII character set.

Function *Compare(VAR first, second: string): boolean*;

This function converts the two input strings to upper-case, then compares them, returning *true* if they are the same, *false* otherwise. The caller of this function should note that the strings are passed by reference, so they will be permanently capitalized.

Procedure *NewLines(count: integer)*;

This simple procedure iterates a loop *count* times, calling a *writeln* each time to print out a new blank line.

Procedure *ModWait(seed: integer)*;

This function is used to give a pseudo-random short delay. The input integer *seed*, presumably something from a random generator, is put into the range 0 to 200 with a call to *mod*, (this is the reason for the function's name), and then a do-nothing *for* loop is executed as many times as the result to give the short delay. One use of this is when we need to get two random numbers at the same time. The first call to *random* will read the system clock, and since the call takes a constant amount of time, the next call to *random* will always return a number with the same relation to the first. If we call *ModWait(seed)* in between, then the first number will have some sort of randomizing effect on the choosing of the second one.

Function *ListLength(List: IList): integer*;

This function finds the length of a partially or fully filled variable of type *IList*. It simply steps through the list until it finds an entry where the *name* field has either "none123", NONE123", or None123". This is our pre-defined stopper value, and is put into the database.

Procedure *BuildString(VAR FinalString: string; NewChar: char)*;

This procedure is used to build up a string one character at a time. It is used in *TimeRead*, where we have to convert a stream of incoming variables of type *char* to one *string*. It is called once for each new character. The string being built is passed into *FinalString*, and the new character to be appended to the end is passed into *NewChar*. This procedure allows the backspace key to be

used as normal. It will delete one character off the end, and will write out the backspace to the screen.

Function *TimeRead(VAR result: string): integer*;

This is used to simulate a Pascal *readln*, to be used where some indication of the player's response time is needed. *TimeRead* reads input as a stream of characters, passing them one at a time to *BuildString* with parameter *result*. This means that at the end of execution of *TimeRead*, the reference parameter *result* will contain the entire string.

When *TimeRead* is first called, the internal clock is read with a call to the library function *Time*. At *EOLN* (end of line), or after two characters have been typed, the time is again read. The difference *LowStop - LowStart* is the number of clock-ticks it took to type two characters, or to type the Return key. The high-order bits of the clock, *HighStart* and *HighStop* are ignored here.

It was found that once in a while the clock would start high, count to *MaxInt*, and start at *negative MaxInt* before being read again. This is checked for and taken care of by the last *if-else* statement.

Procedure *RemoveBlanks(VAR string1: string)*;

If the string parameter *string1* has any trailing *blanks* or *return* characters, they will be removed by this function. White-space not at the end of *string1* will not be removed.

Procedure *Strip(VAR string1: string)*;

This removes all non alpha-numeric characters from *string1*.

Function *CheckAnswer(VAR answer: string; Possibles: NewRec): boolean*;

A *NewRec*, declared in the file *ItemFiler3*, is a record of one database element, or one element in the *InfoList*. Among other things, it contains an array field called *NewRec.names*. Each element of this array is a possible correct answer for the particular item associated with *NewRec*. *CheckAnswer* capitalizes both the string-to-be-checked *answer*, and the possible names found in *Possibles*. All non-alphanumeric characters are also removed. If *answer* matches any one of the names found in *Possibles* or a concatenation of the two names in either order, then *CheckAnswer* returns *true*, otherwise *false*.

8. The File ItemFiler3

Itemfiler3 is another game module (*unit* in UCSD Pascal) that is linked with the main game program *G/.IVAN*. *Itemfiler3* contains procedures and variable declarations that are vital to running of the game. In a nutshell, the primary role of *Itemfiler3* is to interface between the game program and the upper disk drive which contains games images and information vital to the game. It also has made variable declarations that are globally accessible to *G/.IVAN*. These variables most generally have to do with the dynamics of graphics manipulations in the game.

IMPORTANT NOTE: This module is highly DEVICE DEPENDENT since the many graphics procedures and variables defined here are designed specifically for use on the TERAK. If you plan on transferring FLASH IVAN to another machine, it is likely that most of the alterations in the FLASH IVAN game code will most likely occur in this module. Because ITEMFILER3 is highly susceptible to future alterations, we have described variables and procedures in greater depth than we have elsewhere.

8.1. Important Constants

MAXINDEX = 100 :

MAXINDEX indicates the upper bound of the array *INFO_LIST* described below.

MAXNAMES = 3 :

MAXNAMES sets the array in the record defined below to a range of 3.

8.2. Important Global Variables

INFO_LIST : This is an array of records, each of which has a one to one correspondence to an image in the game. Although explained briefly in the GAMEUN3 module, we go into greater detail here since this is where it has been originally declared. Each record is structured as follows:

```
type NEWREC = packed record
    NAMES:       packed array[1..MAXNAMES] of str15;
    BLOCK:       integer;
    FULLSCREEN:  boolean;
    TOPTHIRD:    boolean;
    MIDTHIRD:    boolean;
```

BOTTHIRD: *boolean*;

end;

Note *MAXNAMES* equals 3; *Names[1]* and *Names[2]* hold the game names (up to 15 characters) of a particular image. *Names[3]* holds the fotofile name on which the image is located. *BLOCK* is the block number which corresponds to the fotofile name; after an image directory has been created its fotofile block location on the upper disk is automatically stored in *BLOCK*. By accessing a fotofile be *BLOCK* number using UNITREAD a game image can be accessed 3 to 4 times faster than if it were accessed by name using the usual Pascal file I/O. This makes for a faster, more interesting game. The remaining fields in the record, *FullScreen*, *TopThird*, *MidThird*, and *BotThird* are set to TRUE or FALSE depending upon which part of a fotofile an image is located. These fields enable the game creator the option of putting up to 3 images on fotofile, thus saving disk space. Record indices (in *INFO_LIST*) 90-100 have been set aside for gaming system images such as the opening Eagle animation. Record indices 1-89 are reserved specifically for the actual game images (of Aircraft in the prototype game).

CROSSHAIRS : is of type packed array[0..59,0..59] of boolean. It is a graphics buffer which holds the targetor icon which is flashed to the screen just before a game image is flashed to the screen.

SMALLSC : is a packed array[0..28351] of boolean used as a graphics buffer which is roughly the size of one third the screen. This buffer is used in the instance of a game image stored on a specific third of a *fotofile* which must be flashed to the screen. *SMALLSC* is actually larger than a third of a screen of bits (80 X 320) because it needs to accommodate a UNITREAD call which uses a BLOCK type format in reading information from disk. Thus the size of *SMALLSC* is exactly 7 blocks long (28352 bits). It is stored in a one dimensional array for the sake of convenience and clear understanding. When an image is read into *SMALLSC*, it is read in as a linear string of bytes. In addition to this complication, the very beginning of a thirdscreen image won't always begin at the beginning of the buffer *SMALLSC*, since UNITREAD which begins reading from a Block number cannot always start reading from an exact bit location where an image's string of bytes begins on disk. In the case of a *TopThird* image, there is no problem in this case, since its *Block* number corresponds exactly to its starting bit location. However, in the instance of *MidThird* or *BotThird* images whose starting bit

locations do not exactly corrspond to a BLOCK number on disk, they are UNITREAD from a block number before their starting position. Although they easily fit into the oversized *SMALLSC* buffer, they do not begin at the beginning of *SMALLSC*. By keeping *SMALLSC* one-dimensional, offsets to the beginning of an image in *SMALLSC* are made easy to calculate.

SCREEN : is a packed array[0..239,0..319] of boolean; *SCREEN* acts as a graphics buffer with each boolean element mapping to a particular pixel on the TERAK screen. If a game image is of type *FullScreen*, it is read directly into *SCREEN*. If not, the game image is first read into *SMALLSC* and then bitmapped and centered on *SCREEN*. Whatever *SCREEN* contains can be projected to the screen with the command *UnitWrite(3,SCREEN,63)*. *SCREEN* can be removed from the screen using the command *UnitWrite(3,SCREEN,7)* or *Page(OUTPUT)*.

CLOCK_INT : is a case variant record which is used in procedure *PAUSE*.

HI_LIST : is an array of ten records containing the names and scores of the ten top scoring players. This information is read in from the upperdisk (the images disk) into this array and compared to the score of the current player at the end of each game. If the current player's score is within this range of scores, a new *HI_LIST* is created with his name and score inserted in the appropriate spot and written back to disk. See procedure *HISCORE*.

8.3. Procedures in *ItemFiler3*

Procedure *Delay(N : integer);*

Delay is a simple procedure used to create time delays. A FOR loop is simply executed *N* times. It is estimated that the number of seconds of delay is equal to *N / 100*. So for example *Delay(400)* will simulate a 4 second delay.

Procedure *FromDisk(Var II_list : ILIST; I_name : str15);*

FromDisk reads the image directory (in the case of our game, "#5:NEWNAMES") from the upper disk into the array INFO_LIST.

Procedure *Display(N : integer);*

Display will display the *N*th image in the array *INFO_LIST* on the TERAK screen and leaves it there. It is up to calling program to remove it from the screen; we reasoned that this implementation gives the calling program more freedom as to how long the image is to be displayed. *Display* is very much device dependent, i.e. it is designed to run specifically on the TERAK. This is also true for the entire ITEMFILER3 module. If you plan on transferring FLASH IVAN onto a machine other than the TERAK, *Display* is the procedure that will more than likely need revamping. Because of this procedure's importance, we have outlined it in greater detail than we have other routines:

-will first clear the screen	PAGE(OUTPUT)
-then load the screen buffer SCREEN with *on* bits	FILLCHAR(...)
-then bitmap CROSSHAIRS onto SCREEN	PAINTBLOCK(...)
-then turn on SCREEN	UNITWRITE(...)
-for a second	DELAY(200)
-then turn off screen	UNITWRITE(...)
-then will read the FOTOFILE that the image is	
-- on from disk	UNITREAD(...)
-IF FULLSCREEN = TRUE reads directly to SCREEN	
-ELSE reads to *SMALLSC*	
-- and then offset and bitmapped to SCREEN	
-- then the image is projected to the screen	

Function *Pause : boolean;*

Pause is currently not used in the game and is therefore commented out. Its function is to wait at most 10 seconds for a user response. If a user responds within 10 seconds, control is immediatedly returned to the calling program and *Pause* returns FALSE. If a user hasn't responded within 10 seconds, *Pause* returns TRUE.

Note : uses the case variant record *Clock_Int* described above

Procedure *HiScore;*

First, reads in top ten scores from file "#5:HISCORE.DATA" (on upper disk) and stores them in array *Hi_List.* Next, Displays *HiScore* graphic on the screen (*Info_List[100]*). Next, inserts and sorts current score *SCORE* with scores in *Hi_List*; then outputs *Hi_List* array to the screen, and finally writes the modified *Hi_List* back to disk.

9. The Disks

The Flash Ivan Gaming System consists of three essential disks:

1. The Game Code Disk
   - goes in bottom disk drive
   - holds the game code
   - holds the stats file
2. The Images Disk
   - goes in the upper disk drive
   - holds game *specific* information:
     Fotofiles,
     the image directory,
     instructions,
     and the *HISCORE.DATA* file
3. The Administration Disk
   - goes in the bottom disk drive
     operating on either of the
     other two disks in the top drive
   - used to make a new game
   - and to access and format game stats

10. Organization

With so many different disks floating around, some on the test site and some being used to make game enhancements, we have realized the necessity for tight organization among us. As we have worked on the program, we have adopted three important conventions to better organize ourselves. Firstly, we have chosen the following naming system:

"TOMCAT"[*n*]	corresponding to game code disks
"IVAN_UP"[*n*]	corresponding to images disks
"ADM"	corresponding to the administration disk

where *n* represents a number. Secondly, we keep an exacting written record of each disk: the version of the game on it, where it is, and other vital information. Thirdly, we have designated master disks holding the most recent game enhancements:

"BIGBIRD"	holds most recent game code
"FOTOS1"	holds the most recent image directory, "NEWNAMES" as well as Fotofiles 1-30
"FOTOS2"	holds the remainder of the Fotofiles, Instructions, Initial Hiscores file, etc.

We have made these conventions for our own organizational purposes; you may or may not want to follow them exactly depending on your own tastes.

11. Disk Handling

In the suggestions that follow, we separately discuss the preparations needed for each disk in the FLASH IVAN Gaming System before and after it goes to the gaming site.

11.1. The Game Code Disk -- Before

The game code disk should have a minimal number of files on it before collecting performance data. The two most important files are *SYSTEM.STARTUP*, containing the game code, and *GAMES.DATA*, storing game stats. In order to accommodate new stats written to disk after each game, the game code disk should be Krunched (see P-System details) with *GAMES.DATA* as the last file on the disk. This will allow the statistics file to utilize the remaining disk space the most efficiently. Lastly, it should be checked for any bad blocks.

11.2. The Game Code Disk -- After

When a game code disk returns from collecting data at a testing site, our primary interest is to access the statistics file *GAMES.DATA* and then format it into something readable. The following sequence of instructions make this tedious task less burdensome:

1. Put "ADM" Disk in lower disk drive
2. Put game code disk in upper disk drive
3. Check game code disk for bad blocks
4. Get listing of game code disk.
   -- How big is *GAMES.DATA*?
   -- Is it still the last file on disk?

5a. Be sure *NEWNAMES* is on ADM disk

5b. Clear ADM disk of any unnecessary files
   -- most notably old *CONFIDENCE.TEXT* and *LATENCY.TEXT* files

6. Krunch ADM disk
7. Execute Makestats (be patient; it takes a while to complete)
8. Check to see if new *CONFIDENCE.TEXT* and *LATENCY.TEXT* files are on ADM disk
   -- then print them out
9. If you wish to save these files,
   transfer then to the STATS disks
10. Erase them from the ADM disk
11. Run game code disk with image disk in upper drive
    -- selecting the version option at the beginning,
    see if the version is up to date

11.3. The Images Disk -- Before and After

The Images disk should be checked for bad blocks, frequently. Since data is constantly read from this disk during game time, it absorbs a lot of wear and tear. If ever you Krunch the disk or make any changes to it whatsoever, it is

extremely important that you "convert" the disk afterwards. The "conversion" function can be called from the *MAKEDIR* program on the "ADM" disk and is described in detail in the ***Flash Ivan Utilities*** Documentation.

References

Bowles, K. L. (1977). *Microcomputer problem solving using PASCAL*. New York: Springer-Verlag.

Grogono, P. (1980). *Programming in PASCAL*. Reading MA: Addison-Wesley.

SofTech (1978). *UCSD PASCAL Version II: A product for mini- and microcomputers*. San Diego CA: SofTech Microsystems.

Wagener, J. L. (1980). *FORTRAN 77: Principles of Programming*. New York: John Wiley & Sons.

Appendix A: Program Listing

```
{$S+}

PROGRAM Flashivan;

uses  {$U ITEMFILER3.CODE} ItemFiler3, {$U GAMEUN3.CODE} GameUn3;

CONST
    DateMain = 'Nov 13, 1984... fix "ANSWER", and keyword loop';
    NAME         = 'GAMES.DATA';
    PRACSTART = 91;
    NUMBERPICTURES = 89;
    ORD_ESC = 27;

TYPE
     nametype = string[15];
     sstype = string[11];
     scale = 0..100;

     gamestats = record
        name          : nametype;
        SS            : sstype;
        date          : nametype;
        latency       : packed array[1..numberpictures] of integer;
        confidence    : packed array[1..numberpictures] of scale;
        correct       : packed array[1..numberpictures] of boolean;
     end;

VAR
     TotalShown,gameloop            : integer;

     average,score,total_lat,
     total_conf                     : real;

     scoresfile                     : file of gamestats;
     Minifoto                       : packed array[0..319,0..26] of boolean;
     currentgame                    : gamestats;

{*************************************************************************}
procedure PAINTBLOCK(VAR SOURCE; SRCWID, SRCX, SRCY : INTEGER;
              VAR DEST; DSTWID, DSTX,DSTY,CNTX,CNTY,MODE,GRAY:INTEGER);
      EXTERNAL;

{Assembly-language sound routines...}
procedure CLICK2; EXTERNAL;

procedure TIMEPI; EXTERNAL;

procedure ERROR; EXTERNAL;


procedure afterpicture(right:boolean;conf:integer;lat,index:integer);
                  FORWARD;
procedure SingleTrial(ListIndex:integer; ifprac : boolean);
                  FORWARD;
```

```
procedure OutputStats; FORWARD;

{*****************************************************************************}


SEGMENT PROCEDURE ANIMATE;
VAR XZERO,YZERO : INTEGER;
BEGIN
  UNITWRITE(3,SCREEN,7);  { DON'T SEND SCREEN TO CRT }
                    {PAINT IN TO THE MINIFOTOS}
  UNITREAD(5,SCREEN,SIZEOF(SCREEN),INFO_LIST[91].BLOCK);
  PAINTBLOCK(SCREEN,320,0,0,MINIFOTO,320,0,0,320,26,0,-1);

  PAINTBLOCK(SCREEN,320,0,30,CROSSHAIRS,60,0,0,60,60,0,-1);
                    { FLASH IVAN }
  { PAINTBLOCK(SCREEN,320,0,120,SCREEN,320,0,130,320,60,0,-1); }

    {  TITLE SEQUENCE STARTS HERE }
   {  DISPLAY EAGLE }
  PAGE(OUTPUT);
  UNITREAD(5,SCREEN,SIZEOF(SCREEN),INFO_LIST[90].BLOCK);
  UNITWRITE(3,SCREEN,63);

  UNITREAD(5,SMALLSC,SIZEOF(SMALLSC),INFO_LIST[91].BLOCK+6);
  XZERO:=64; YZERO:=3;  { PLACE IN SMALLSC WHERE THIRDSCREEN REALLY STARTS}

                    { EAGLE ANIMATION1}
  PAINTBLOCK(SMALLSC,320,XZERO,YZERO+15,SCREEN,320,101,50,100,55,0,-1);
  UNITWRITE(3,SCREEN,63);
  DELAY(50);              { EAGLE ANIMATION2}
  PAINTBLOCK(SMALLSC,320,XZERO+100,YZERO+15,SCREEN,320,101,50,100,55,0,-1);
  UNITWRITE(3,SCREEN,63);
  DELAY(50);
                    { EAGLE ANIMATION3}
  PAINTBLOCK(SMALLSC,320,XZERO+200,YZERO+15,SCREEN,320,101,50,100,55,0,-1);
  UNITWRITE(3,SCREEN,63);

  TIMEPI;
            { EAGLE ANIMATION2}
  PAINTBLOCK(SMALLSC,320,XZERO+100,YZERO+15,SCREEN,320,101,50,100,55,0,-1);
  UNITWRITE(3,SCREEN,63);
  DELAY(50);              { EAGLE ANIMATION1}
  PAINTBLOCK(SMALLSC,320,XZERO,YZERO+15,SCREEN,320,101,50,100,55,0,-1);
  UNITWRITE(3,SCREEN,63);

  UNITREAD(5,SMALLSC,SIZEOF(SMALLSC),INFO_LIST[91].BLOCK+12);
  XZERO:=128; YZERO:=6;
  PAINTBLOCK(SMALLSC,320,XZERO,YZERO+5,SCREEN,320,0,121,320,65,0,-1);
  UNITWRITE(3,SCREEN,63);
END;


{
```

```
*******************************
 SEGMENT PROCEDURE PRACTICE
*******************************
}

SEGMENT PROCEDURE PRACTICE;

VAR
      i, fx, millisecs, FakeList,
      ListIndex, browse_index,
      confidence                    : integer;

      correct                       : boolean;
      resp                          : char;
      resp_string                   : string;

 procedure INSTRUCT;

 Var
          I_String    : string;
          directions  : text;
          i           : integer;
          resp        : char;
 BEGIN
    page(output);
    reset(directions, '#5:INSTRUCT.TEXT');
    while not EOF(directions) do
    begin
     for i := 1 to 21 do
     begin
      if not EOF(directions) then
      begin
        readln(directions, I_String);
        writeln(I_String);
      end;
     end;
     writeln('HIT <RETURN>');
     readln;
    end;
    close (directions);
 END;    { INSTRUCT }


 procedure PLAY3;
 begin
  PAGE(OUTPUT);
  GOTOXY(29,8); WRITE('Be Prepared to Answer');
  GOTOXY(28,9); WRITE('the following 3 examples');
  GOTOXY(17,11); WRITE('WATCH THE CENTER OF THE SCREEN FOR AIRCRAFT');
  GOTOXY(20,14); WRITE('Hit  <RETURN>  When  Ready  To  Begin');
  READLN;
  for ListIndex := 95 to 97 do  { Loop over practice pictures }
  begin
   FAKELIST := FAKELIST + 1;
```

```
    Display(ListIndex);
    SingleTrial(FAKELIST,true);
    page(output);
  end;
end; { PLAY3 }


procedure ANSWER( VAR A1 : STRING; X, CONF_KEY : INTEGER );
begin
  DELAY(2500);PAGE(OUTPUT);
  GOTOXY(32,2);
  FILLCHAR(SCREEN,SIZEOF(SCREEN),0);
              { AIRCRAFT }
  PAINTBLOCK(MINIFOTO,320,0,0,SCREEN,320,0,22,121,6,0,-1);
  UNITWRITE(3,SCREEN,63);
  DELAY(2000);
  FOR I := 1 TO X DO        { AUTOMATE RESPONSE TO PROMPT }
  BEGIN
    CLICK2;CLICK2; WRITE(A1[I]);
    DELAY(70); CLICK2;CLICK2; DELAY(30);
  END;
  DELAY(1500);
              { CONFIDENCE RULER }
  GOTOXY(2,5); WRITE('LEAST');
  GOTOXY(0,6); WRITE('CONFIDENT');
  GOTOXY(73,5); WRITE('MOST');
  GOTOXY(70,6); WRITE('CONFIDENT');
  PAINTBLOCK(MINIFOTO,320,0,6,SCREEN,320,0,75,320,20,0,-1);
  UNITWRITE(3,SCREEN,63);
              { GIVE CONFIDENCE PROMPT}
  PAINTBLOCK(MINIFOTO,320,125,0,SCREEN,320,0,111,135,6,0,-1);
  UNITWRITE(3,SCREEN,63);
  GOTOXY(32,11);
  DELAY(3000);
  CLICK2;CLICK2;
  WRITE(CONF_KEY);
  DELAY(100);
  CLICK2;CLICK2;
  DELAY(500);
  WRITELN;
  (********** Simlulates SingleTrial **********)
  FAKELIST := FAKELIST + 1;
  millisecs := 1556;
  correct := TRUE;
  if (conf_key = 0) then confidence := 100
    else confidence := conf_key * 10;

  TotalShown := TotalShown + 1;
  flashscore := flashscore + 1;
  gotoxy(0,14);
  writeln('RECOGNITION CORRECT.');
  writeln('Response Time = ',millisecs/1000:4:2,' seconds');
  afterpicture(correct,confidence,millisecs,FAKELIST);
  flashtotal := flashtotal + 1;
```

```
  OutputStats;
  writeln;
  write('   < Hit RETURN for next Aircraft >');

  READLN;
  PAGE(OUTPUT);
 END { ANSWER };


{ ———————— Driver for PRACTICE ————————————————}

BEGIN     { PRACTICE }
 gotoXY(26,5);
 write('HIT RETURN TO BEGIN GAME');
 resp_string := '';
 resp := 'x';
 reset(keyboard);
 readln(resp_string);

 if (resp_string = 'boatman') then
 begin

 { while not <ret>, <space>, or <esc>... (UCSD Pascal returns the same
  character for <ret> and <space>)                                    }

  while ((resp <> chr(32)) and (resp <> chr(27))) do
  begin
   case ord(resp) of

     (********
     100, 68 {'d', 'D' ...display a picture} :
      begin
       page(output);
       writeln('What picture do you want to see? (give index number) ');
       readln(browse_index);
       if ((browse_index > 0) and (                Display(browse_index);
        writeln(info_list[browse_index].Names[1]);
        writeln(info_list[browse_index].Names[2]);
       end
       else
        writeln('Invalid index number. ');
      end;
     ******)

     104, 72 {'h', 'H' ...show the High Scores} :
      begin
       page(output);
       HiScore('', 0);
      end;

     109, 77 {'m', 'M' ...call memavail() } :
      begin
       page(output);
       write('The memory available in segment procedure Practice is ');
```

```
          writeln(memavail, ' words.');
        end;

      118, 86 {'v', 'V' ...show the current dates of the game files} :
        begin
          page(output);
          writeln('G/.Ivan version of ', DateMain);
          writeln('ItemFiler3 version of ', DateItemFiler3);
          writeln('GameUn3 version of ', DateGameUn3);
        end;

    end; { ...of case... }

    writeln;
    write('Hit <ret> to go on ');
    read(resp);

  end { ...while... };

end { ...if not keyword... };


if (resp <> chr(27)) then        { not an <esc>, so show all instructions... }
  begin
    INSTRUCT;
    PAGE(OUTPUT);
    GOTOXY(27,7); WRITE('Be Prepared to Observe');
    GOTOXY(25,8); WRITE('3 Automated Game Examples');
    GOTOXY(25,10); WRITE('Hit <RETURN> to Continue.');
    READLN;

    confidence := 0;
    FAKELIST := 0;

    DISPLAY(92);
    ANSWER(INFO_LIST[92].NAMES[2],4,6);

    DISPLAY(93);
    ANSWER(INFO_LIST[93].NAMES[2],13,0);

    DISPLAY(94);
    ANSWER(INFO_LIST[94].NAMES[2],5,1);

    PLAY3;

  end;

END;   { FRONT END }


{*****************************************************************************
 SEGMENT PROCEDURE HELLO  composed of the following:

 Shows the player the "privacy act" (text written into a fotofile), gets
```

```
the player's name (making sure it is 15 characters or less), gets the
Social Security # (and makes sure it is 11 characters or less), and gets
the date.  It then shows it's results to the player, allowing changes.
*************************************************************************}

SEGMENT procedure HELLO(var player:nametype;var date:{date}nametype;
                        var SS:sstype);
var str : string;
    resp : char;
    socsec : nametype;


  procedure getdate(var date: {date}nametype);
    var
       str : string;
  begin
    str  := '';
      repeat
        if length(str) > 0 then writeln('DATE IS TOO LONG ');
        WRITE('DATE  ( Day, Month, Year )  : ');
        readln(str);
      until ((length(str)>0) and (length(str)<16));
      date  := str;
  end;



begin {of Hello-- main body
  PAGE(OUTPUT);
  UNITREAD(5,SCREEN,SIZEOF(SCREEN),INFO_LIST[98].BLOCK);
  UNITWRITE(3,SCREEN,83);
  GOTOXY(0,30);
  WRITELN('Hit <RETURN> for next page.');
  READ(RESP);
  PAGE(OUTPUT);
  UNITREAD(5,SCREEN,SIZEOF(SCREEN),INFO_LIST[99].BLOCK);
  UNITWRITE(3,SCREEN,83);
  GOTOXY(0,30);
  WRITE('HIT <RETURN> WHEN DONE.');
  READ(RESP);
  repeat
    page(output);
    str  := '';
    repeat
      if length(str) > 0 then
        writeln('The name is too long.  It must be 15 characters or less. ');

      write('NAME (Last name, first initial, middle initial) :');
      readln(str);
    until ((length(str)>0) and (length(str)<16));

    player := str;

    str  := '';
```

```
    repeat
      if length(str) > 0 then writeln('Social security number is too long.');
      write('SOCIAL SECURITY # :');
      readln(str);
    until ((length(str)>0) and (length(str)<12));
    SS  := str;
    GetDate(date);
    writeln;{ writeln;}
    writeln(' Is this correct? :');
    writeln;
    writeln('NAME          : ',player);
    writeln('S.S. NUMBER   : ',SS);
    writeln('DATE          : ',date.day,'-',date.month,'-19',date.year});
    writeln;
    write('Is this correct?  [y or n] ');
    read (resp);
  until (resp in ['y', 'Y']);

  PAGE(OUTPUT);
  GOTOXY(26,7); WRITE('Are You Ready to Play');
  GOTOXY(32,8);WRITE('FLASH IVAN ?!');
  GOTOXY(24,10);WRITE('If So, Then Hit <RETURN>.');
  GOTOXY(17,12); WRITE('WATCH THE CENTER OF THE SCREEN FOR AIRCRAFT');

  GOTOXY(33,15); WRITE('Good Luck!!!');
  READLN;
end;



{*****************************************************************************
 PROCEDURE AFTERGAME is run after every game and records the player's game to
 the disk under the file name "NAME" (Games.data). This is a file of type
 gamestats.
*****************************************************************************}
procedure aftergame;
var iscore : integer;
begin
  iscore := trunc(score);
  page(output);
  gotoxy(0,5);
  write('            YOUR FINAL SCORE :  ',iscore :6,' pts.');
  gotoxy(0,15);
  writeln('             YOUR FINAL RESULTS:');
  writeln;
  OutputStats;
  writeln;
  write('   Hit <RETURN> for Hi-SCORES');
  readln;
  HISCORE(currentgame.name,iscore);
  reset(scoresfile,NAME);

  repeat
    get(scoresfile);
```

```
  until eof(scoresfile);

  scoresfile^ := currentgame;
  put(scoresfile);
  close (scoresfile,lock);
end;


{*****************************
  procedure OutputStats-  outputs statistical data concerning game performance
                          in PERCENTAGE CORRECT, AVERAGE CONFIDENCE, and
                          AVERAGE-LATENCY;
 *****************************}
procedure OutputStats;              (* Output to the User *)
begin

  {GOTOXY(32,16);}
  writeln('PERCENT CORRECT RECOGNITIONS :  ':66,
  round(average), ' / ', TotalShown, ' ', 100*average / TotalShown:5:1,'%');

  {GOTOXY(32,17);}
  writeln('AVERAGE RECOGNITION CONFIDENCE :  ':66,
  total_conf / TotalShown:6:1,'%');

  {GOTOXY(32,18);}
  writeln('AVERAGE RESPONSE TIME : ':65,
  round(total_lat / TotalShown)/1000:4:2,' seconds');
END;


{*****************************
  procedure InitStats  -  initializes GLOBAL variables for statistical purposes
 *****************************}
procedure InitStats;
begin
  flashscore := 0;
  flashtotal := 0;
  TotalShown := 0;
  score := 0;
  average    := 0;
  total_lat  := 0;
  total_conf := 0;
  for x := 1 to numberpictures do
    currentgame.latency[x] := 0;
end;


{*******************************
  Procedure AfterPicture
 *******************************}
```

```
procedure afterpicture;  { SEE above for parameter list }
var x : integer;
   r,rscore,rconf,rlat : real;
begin
  with currentgame do
  begin
    latency[index] := lat;
    total_lat := total_lat + lat;
    confidence[index] := conf;
    total_conf := total_conf + conf;
    correct[index] := false;
    if right then
    begin
      correct[index] := true;
      average := average + 1;
    end;
    r := 1.0;
    rconf := conf/10; rlat := lat*r;
    if right then
       score := score + ((rconf*1030)/(1000+rlat)) + 10
      else
       score := score - ((rconf*1030)/(1000+rlat)) - 10;
  end;
end;



{*************************
  procedure GetConfidence...

   This procedure will prompt for the user's own confidence rating, read it
   as a character, convert it to an integer, and send it back as the VAR
   parameter "conf"
 *************************}

procedure GetConfidence(VAR conf: integer);
var
      c_response : char;
begin

                  { CONFIDENCE RULER }
    GOTOXY(2,5); WRITE('LEAST');
    GOTOXY(0,6); WRITE('CONFIDENT');
    GOTOXY(73,5); WRITE('MOST');
    GOTOXY(70,6); WRITE('CONFIDENT');
    PAINTBLOCK(MINIFOTO,320,0,6,SCREEN,320,0,75,320,20,0,-1);
    UNITWRITE(3,SCREEN,63);
                  { GIVE CONFIDENCE  }
    PAINTBLOCK(MINIFOTO,320,125,0,SCREEN,320,0,111,135,6,0,-1);
    UNITWRITE(3,SCREEN,63);
    GOTOXY(32,11);
    read(c_response);                  { read response as a string }
```

```
    {*** convert the string to an integer... ***}
    while not (ord(c_response) in [9,48..57]) do
     begin
       GOTOXY(0,12);
       WRITELN('     You must enter a number from the above set...');
       WRITE('CONFIDENCE : ');
       read(c_response);
       writeln;
     end;
    if (c_response in ['0'..'9']) then
    begin
      conf := (ord(c_response) - ord('0'));
      if (conf = 0) then conf := 100
        else conf := conf * 10;
    end
    else conf := 0;
end { ...of procedure GetConfidence... };



{*****************************************************************************}
Procedure SingleTrial;  {See Parameter list above }
VAR   j,ticks,millisecs,confidence  : integer;
                          time  : real;
                  correct,got   : boolean;
                        guess   : string;
                        resp    : char;
 begin
   TotalShown := TotalShown + 1;
   for j := 1 to SetSpeed do DELAY(250);
   page(output);
     { AIRCRAFT NAME PROMPT }
   GOTOXY(32,2);
   FILLCHAR(SCREEN,SIZEOF(SCREEN),0);
   PAINTBLOCK(MINIFOTO,320,0,0,SCREEN,320,0,22,121,6,0,-1);
   UNITWRITE(3,SCREEN,63);
   ticks := TimeRead(guess);
   if (guess = '') then guess := 'XXXXX';       { to insure wrong answer    }

   IF IFPRAC THEN
     CORRECT := CHECKANSWER(GUESS,INFO_LIST[PRACSTART+LISTINDEX])
   ELSE
   begin
     correct := CheckAnswer(guess, Info_List[ListIndex]);
   end;


   GetConfidence(confidence);        { read the user's confidence          }
      { Calculate the latency to answer in seconds, and milliseconds... }
   time := (ticks/60);
   if (time > (MaxInt/1000)) then time := (MaxInt/1000);
      { ...so we don't get an overflow when converting to milliseconds... }
   millisecs := round(time * 1000);
```

```
WRITELN;
writeln;
IF (correct) THEN
 BEGIN
  flashscore := flashscore + 1;
  GOTOXY(0,14);
  WRITELN('RECOGNITION CORRECT.');
 END
ELSE
 BEGIN
  GOTOXY(0,14);
  WRITELN('RECOGNITION INCORRECT.');
  IF IFPRAC THEN BEGIN
   WRITE('That was a ',Info_List[PRACSTART+ListIndex].Names[2]);
   WRITELN(' ',Info_List[PRACSTART+ListIndex].Names[1]);
  END
  ELSE BEGIN
   WRITE('That was a ',Info_List[ListIndex].Names[2]);
   WRITELN(' ',Info_List[ListIndex].Names[1]);
  END; { IFPRAC }
  ERROR; { sound for bad response }
 END;
writeln('Response Time = ',millisecs/1000:4:2,' seconds');
afterpicture(correct, confidence, millisecs, ListIndex);
flashtotal := flashtotal + 1;
OutputStats;
writeln;
write('   Hit <TAB> to see the aircraft again, ');
writeln('<RETURN> for next Aircraft ');

read(resp);

if resp <> chr(9)              { The TAB key...              }
then begin
 readln;                       { Eat up the "return"...      }
end
else
begin
 if (IFPRAC) then
 begin
  Display(PRACSTART + ListIndex);
  writeln(info_list[PRACSTART + ListIndex].Names[2]);
  writeln(' ', info_list[PRACSTART + ListIndex].Names[1]);
  writeln;
  write('Hit <RETURN> for next aircraft ');
  readln;
 end
 else
 begin
  Display(ListIndex);
  writeln(info_list[ListIndex].Names[2]);
  writeln(' ', info_list[ListIndex].Names[1]);
  writeln;
  write('Hit <RETURN> for next aircraft ');
```

```
      readln;
    end;

  end;

  {************   for 10 second max wait
  unitread(2,resp,1,0,1);
  got := PAUSE;
  if got=false then begin
    ERROR;
    PAGE(OUTPUT);
    GOTOXY(33,10); WRITE('TIME EXPIRED');
    GOTOXY(15,12); WRITE('Watch the center of the screen for next aircraft');
    DELAY(7000);
  end;
  ****************}

 end { SingleTrial };



{ ********************************************************************************** }
PROCEDURE Game2;
VAR ListIndex,PicLoop  : integer;
begin
 page(o  put);
 for P.cLoop := 1 to TotalPictures do  { Loop over the entire set of pictures }
 begin
   ListIndex := PicSequence[PicLoop];  { get the next index from the random
                                         ordering                        }
   Display(ListIndex);
   SingleTrial(ListIndex,false);
   page(output);
 end;
 score := (score + 838.78)*(1000.0/1677.56);
end { Game2 };



BEGIN                        { MAIN PROGRAM }
 gameloop := 0;
 while gameloop = 0 do begin
   InitStats;
   FromDisk(Info_List, '#5:NewNames');
   ANIMATE;
   TotalPictures := (**** ListLength(Info_List) ****) 42;
     ChoosePlanes(PicSequence);
   Shuffle(PicSequence, TotalPictures);
   PRACTICE;
   HELLO(currentgame.name,currentgame.date,currentgame.SS);
      { corbin's GETNAME,GETDATE, confirm in a gift rap }
```

```
    InitStats;
    Game2;
    AfterGame;
    Readln;
  end;
END.
```

```
{$S+}
UNIT ITEMFILER3;

{ ********************************************************************************* }

INTERFACE

{ ********************************************************************************* }

CONST
     DateItemFiler3 = '10-9-84... HiScores no longer asks for "call sign"-- G
     MAXINDEX = 100;
     MAXNAMES = 3;
     FLAGS   = 100;  { Fotofile index number in INFO_LIST }

TYPE  STR15 = STRING[15];
    AIM_PIC = PACKED ARRAY[0..59,0..59] OF BOOLEAN;
    THIRDSCREEN = PACKED ARRAY[0..28351] OF BOOLEAN;  { 7 BLOCKS OF BITS }
    SCREENMAP = PACKED ARRAY[0..239,0..319] OF BOOLEAN;
    CLOCK_INT = RECORD CASE BOOLEAN OF
                 TRUE : (VAL : INTEGER);
                 FALSE : (BOOLS : PACKED ARRAY[0..15] OF BOOLEAN);
                END;
    SCORES_REC = PACKED RECORD
           GAMENAME : STRING[15];
           SCORE    : INTEGER;
          END;
    NEWREC = packed record
           names : packed array[1..MAXNAMES] of str_5;
           block : integer;
           FULLSCREEN :BOOLEAN;
           TOPTHIRD : BOOLEAN;
           MIDTHIRD : BOOLEAN;
           BOTTHIRD : BOOLEAN;
          END;

   ILIST = array[1..MAXINDEX] of NEWREC;

VAR infodir : file of NEWREC;
   HISCOREFILE : FILE OF SCORES_REC;
   SCREEN : SCREENMAP;
   SMALLSC : THIRDSCREEN;
   CROSSHAIRS : AIM_PIC;
   INFO_LIST : ILIST;
   HILIST    : PACKED ARRAY[1..10] OF SCORES_REC;
   I_NAME : STR15;

procedure FROMDISK(Var II_list:ILIST; I_name:str15);

procedure DISPLAY( N : integer );

procedure DELAY( N : Integer );

{procedure MEMORY( M : integer );}
```

```
{function PAUSE : boolean;}

procedure HISCORE(name : str15; score : integer);


{ ************************************************************************}

IMPLEMENTATION

{ ************************************************************************}


Procedure PAINTBLOCK(Var source; srcwid, srcx,srcy:integer;
                var dest; dstwid,dstx,dsty, cntx,cnty,mode,gray:integer);
        (* mode :  0=store, 1=or, 2=and, 3=xor; +4=comp *)

        External;



{procedure MEMORY;
begin
  writeln('PLACE #',M,'  ','MEMAVAIL=',MEMAVAIL,' SIZEOF(infordir)=',
                                        SIZEOF(INFODIR) );
  readln;
end;}

procedure DELAY;
var i : integer;
begin
  for i := 1 to N do;
end;


procedure FROMDISK;
var H : integer;
begin
    reset(infodir,I_name);
    FOR H := 1 TO MAXINDEX DO BEGIN
      II_list[H] := infodir^;
      if not EOF(infodir) then get(infodir);
    end;
    close(infodir);
end;

{ ************************************************************************
  DISPLAY -- Displays a game image on the screen according to it's
             index number in II_list.  qq
             }
procedure DISPLAY;
var NEWBLOCK,X,Y : INTEGER;
begin
```

```
 PAGE(OUTPUT);
 with INFO_LIST[N] do begin
  fillchar(SCREEN,sizeof(SCREEN),255);
  PAINTBLOCK(CROSSHAIRS,60,0,0,SCREEN,320,129,89,60,60,3,-1);
  UNITWRITE(3,SCREEN,63);
  DELAY(200);
  UNITWRITE(3,SCREEN,7);
  IF FULLSCREEN=TRUE THEN BEGIN
   UNITREAD(5,SCREEN,sizeof(SCREEN),BLOCK);
   UNITWRITE(3,screen,63);
  END
  ELSE BEGIN
   IF (TOPTHIRD=TRUE) THEN BEGIN
    NEWBLOCK:=BLOCK;
    X := 0; Y:=0;
   END;
   IF (MIDTHIRD=TRUE) THEN BEGIN
    NEWBLOCK:=BLOCK+6;
    X := 64; Y := 3;
   END;
   IF (BOTTHIRD=TRUE) THEN BEGIN
    NEWBLOCK:=BLOCK+12;
    X := 128; Y := 6;
   END;
   UNITREAD(5,SMALLSC,SIZEOF(SMALLSC),NEWBLOCK);
   PAINTBLOCK(SMALLSC,320,X,Y,SCREEN,320,0,79,320,80,0,-1);
   UNITWRITE(3,screen,63);
  END;
 end; { with }
end; { Display }


{ ***************************************************************************
function PAUSE : boolean;
     Waits  10  seconds for a user response.
     returns control to the program when a response is detected
     or after 10 seconds
     returns a boolean value
*************************************************************************** }
{******************
function PAUSE;
VAR LO,LO1 : CLOCK_INT;
   HI : INTEGER;
BEGIN
 PAUSE := FALSE;
 TIME(HI,LO.VAL);
 LO.BOOLS[0] := TRUE;
 WHILE UNITBUSY(2) DO
  BEGIN
   TIME(HI,LO1.VAL);
   LO1.BOOLS[0] := FALSE;
   IF ((LO1.VAL-LO.VAL)>600) THEN EXIT(PAUSE);
  END;
 PAUSE := TRUE;
```

```
END;
**********************}


PROCEDURE HISCORE;

VAR
    INC, H          : INTEGER;
    DONE            : BOOLEAN;
    c_response      : char;
    str             : string;


begin
 reset(hiscorefile,'#5:HISCORE.DATA');
 for h:= 1 TO 10 DO BEGIN
  HILIST[H] := HISCOREFILE^;
  IF NOT EOF(HISCOREFILE) THEN GET(HISCOREFILE);
 END;
 CLOSE(HISCOREFILE);
 PAGE(OUTPUT);
 DONE := FALSE;
 INC := 0;


 REPEAT
  INC := INC+1;

  IF INC=11 THEN
    DONE:=TRUE
  ELSE
    IF (Hilist[inc].Score <= Score) THEN
      DONE:=TRUE;

 UNTIL DONE;

 IF INC<>11 THEN BEGIN

  IF INC<>10 THEN BEGIN
   FOR H := 10 DOWNTO (INC+1) DO BEGIN
    HILIST[H].GAMENAME := HILIST[H-1].GAMENAME;
    HILIST[H].SCORE := HILIST[H-1].SCORE;
   END;
  END;


  {**************************************
  {This player will be on the list... decide what name to put there.}

  page(output);
  writeln; writeln; writeln; writeln; writeln;
  writeln(' Congratulations.  Your score is one of the ten best so far,');
  writeln(' and will be put on the list.  Would you like to change the');
```

```
write('  name "', name, '" to your "call sign" instead?  [y or n]  ');
read(c_response);

while not (c_response in ['y', 'Y', 'n', 'N']) do
begin
   gotoXY(0, 11);
   write('  Please enter a "y" or an "n":  ');
   read(c_response);
   writeln;
end;

if (c_response in ['n', 'N']) then
begin
  writeln('Okay.  "', name, '" it is.');
end
else
begin
  repeat

    repeat
      page(output);
      writeln;
      write('  Please type in the new name (15 characters or less) : ');
      readln(str);
    until ((length(str) > 0) and (length(str) < 16));

    name := str;

    writeln;
    write('  Is "', name, '" correct? [y or n] ');
    read(c_response);

  until (c_response in ['y', 'Y']);

end; { of "else" }
*******************)

 HILIST[INC].GAMENAME :=NAME;
 HILIST[INC].SCORE := SCORE;
END;

PAGE(OUTPUT);
UNITREAD(5,SCREEN,SIZEOF(SCREEN),INFO_LIST[FLAGS].BLOCK);
UNITWRITE(3,SCREEN,63);

FOR H := 1 TO 10 DO BEGIN
  GOTOXY(29,7+H); WRITE(HILIST[H].GAMENAME);
  GOTOXY(47,7+H); WRITE(HILIST[H].SCORE);
END;
REWRITE(HISCOREFILE,'#5:HISCORE.DATA');
FOR H := 1 TO 10 DO BEGIN
  HISCOREFILE^ := HILIST[H];
  PUT(HISCOREFILE);
END;
```

```
  CLOSE(HISCOREFILE,LOCK);
END;   { HISCORE }

END. { UNIT }
```

```
{$S+}
Unit GameUn3;


INTERFACE
uses {** Menus, **} {$U ITEMFILER3.CODE}ItemFiler3;


CONST
     DateGameUn3 = '7-17-84, 3:12 PM... by a friend of Lethe's Boatman';
     MaxInt      = 32767;
     SetSpeed    = 7;          { Speed for games with no speed option }
     ChooseGame    = True;       { Allow user to choose game          }
     All_In_One    = False;      { Play the game that shows each and
                                   every picture once                 }
     FlashGame { currently not working }
                 = False;      { Play the game that chooses a picture
                                 from the entire set each time        }

TYPE
 TINY_STRING = string[1];
     CharString    = string[1];   { Handy for concat, etc.            }
     NameRec       = record
                     Name1 : string[15];
                     Name2 : string[15];
                     {****
                     Name3 : string[15];
                     Name4 : string[15];
                     ****}
                         { If the number of these fields
                           is changed, procedure CheckAnswer must also
                           be changed.    }
                   end; { ...of record NameRec... }

     {***
     PicList       = array[1..MaxPictures] of NameRec;
     ***}
     IndexList     = array[1..MaxIndex] of integer;    { This will hold
                       a  random ordering of all possible indices. }


 VAR

     TotalPictures  : integer;     { To hold the total number of pictures }
     PicSequence    : IndexList;   { For the game's order to show pictures}
     flashscore     : integer;     { Globally keep track of total correct }
     flashtotal     : integer;     { Globally keep track of total trials  }
     seed               : integer;
     response           : char;
     game           : char;
     done               : boolean;
     x,i                : integer;
     {** menu1           : MenuRecord; **}
     PlaneName          : string[15];
```

```
    PictureTotal        : integer;

PROCEDURE Randomize(VAR seed: integer);

FUNCTION Random(VAR seed: integer; Low, High: integer) : integer;

Procedure Shuffle(VAR IndexArray: IndexList; Num_of_Pics: integer);

Procedure MakeSequence(VAR IndexArray: IndexList; Num_of_Pics: integer);

Procedure ChoosePlanes(VAR IndexArray: IndexList);

PROCEDURE UpperCase(VAR name1 : string);

FUNCTION Compare(var first, second: string): boolean;

PROCEDURE NewLines(count: integer);


PROCEDURE ModWait(seed : integer);

Function ListLength(VAR List: IList): integer;
Procedure BuildString(Var FinalString: string; NewChar: Char);
Function TimeRead(Var result: string): integer;
Procedure RemovePlanks(VAR string1: string);

Procedure S ip(VAR string1 : string);

Function Che kAnswer(var answer: string; Possibles: NewRec): boolean;

(************************************)
IMPLEMENTATION
(************************************)

PROCEDURE Randomize;
  external;


function random;

CONST
      L = 29;
      C = 217;
      M = 1024;

VAR
      fraction        : real;

begin
  {***
  realseed := (abs(seed*27.182813)) + 31.415917;
  realseed := realseed / 100;
  realseed := realseed-trunc(realseed);
```

```
  ***}
  seed := abs(seed) mod 1000;
  seed := (seed * L + C) mod M;
  fraction := seed / M;

  random := trunc((fraction * ((high - low) + 1)) + low);
end;


{-----------------------------------------}
procedure Shuffle;
{  Shuffle the list by randomly interchanging pairs of entries.          }

VAR
      rand1, rand2    : integer;
      i               : integer;
      index1, index2  : integer;
      temp            : integer;


begin
  randomize(rand1);             { start one random sequence                }

  ModWait(rand1);         { wait a random amount of time (to let the
                            clock reach another random state)       }
  randomize(rand2);             { start the other random sequence          }
  for i := 1 to 200 do          { make 200 random exchanges of elements    }
  begin
    index1 := random(rand1, 1, Num_of_Pics);
    index2 := random(rand2, 1, Num_of_Pics);    { randomly choose a pair of
                                                  elements to interchange... }
    temp := IndexArray[index1];

    IndexArray[index1] := IndexArray[index2];
    IndexArray[index2] := temp;                 { ...and interchange them  }
  end { ...of "for" loop... };
end { ...of procedure Shuffle()... };
{-----------------------------------------}

{-----------------------------------------}
Procedure MakeSequence;
{
      This procedure will fill the array IndexArray with a shuffled
    sequence of index values, for use as a random sequence when each
    index should only be used once.
}
VAR
      count1, count2  : integer;

begin
{ First, initialize the array to an ordered sequence. }
  for count1 := 1 to Num_of_Pics do
    IndexArray[count1] := count1;
```

```
  Shuffle(IndexArray, Num_of_Pics);
end; { ...of procedure MakeSequence... }
{----------------------------------------}



{----------------------------------------}
procedure ChoosePlanes;
{
      This procedure will fill IndexArray with 42 index numbers, c
   corresponding to 7 each Russian fronts, sides, and tops, and their
   matching distractors.  This will only work if they are arranged with
   their index values as 14 Russian (same view), 14 distractors, etc.
   The corresponding distractor for any Russian picture should have an index
   of 14 greater.
      IndexList will contain randomly chosen indices, but they will not
   be randomly grouped.  The list variable should be passed to a shuffling
   routine such as Shuffle() after it is filled here.
}

VAR
      rand1          : integer;
      AlreadyChosen  : set of 1..MaxIndex;
      NewNumber      : integer;
      i              : integer;
      check          : integer;

begin
  randomize(rand1);
  AlreadyChosen := [];

  for i := 0 to 6 do                    { get 7 each Russ. and dist. tops... }
  begin
    NewNumber := random(rand1, 1, 14);          { choose a Russian...          }

    check := 0;
    while ((NewNumber in AlreadyChosen) and (check <= 14)) do
    begin
      {***
      NewNumber := ((NewNumber + 1) MOD 14) + 1;
      check := check + 1;
      ***}
      NewNumber := random(rand1, 1, 14);
      write('.');
    end; { ...find an unused number... }

    if (check > 14) then
    begin
      writeln;
      write('ERROR: trouble in procedure ChoosePlanes, cannot find new number.')
      writeln;
    end;

    AlreadyChosen := AlreadyChosen + [NewNumber];
```

```
    IndexArray[(2 * i) + 1] := NewNumber;
    IndexArray[(2 * i) + 2] := NewNumber + 14;  { add corresponding distractor }

  end;

{ Get next view set... }
  randomize(rand1);
  AlreadyChosen := [];

  for i := 7 to 13 do                    { get 7 Russ. and dist. sides  }
  begin
    NewNumber := random(rand1, 29, 42);        { choose a Russian...        }

    check := 0;
    while ((NewNumber in AlreadyChosen) and (check <= 14)) do
    begin
      {***
      NewNumber := (NewNumber + 1) MOD 14;
      if (NewNumber = 0) then NewNumber := 14;
      ***}
      NewNumber := random(rand1, 1, 14);
      write('.');
      NewNumber := NewNumber + 28;             { put in range of 29 to 42...}
    end; { ...find an unused number... }
    if (check > 14) then
    begin
      writeln;
      write('ERROR: trouble in procedure ChoosePlanes, cannot find new number. )
      writeln;
    end;

    AlreadyChosen := AlreadyChosen + [NewNumber];

    IndexArray[(2 * i) + 1] := NewNumber;
    IndexArray[(2 * i) + 2] := NewNumber + 14;  { add corresponding distractor }
  end;


{ get next view set... }
  randomize(rand1);
  AlreadyChosen := [];

  for i := 14 to 20 do
  begin
    NewNumber := random(rand1, 57, 70);        { choose a Russian...        }

    check := 0;
    while ((NewNumber in AlreadyChosen) and (check <= 14)) do
    begin
      {***
      NewNumber := ((NewNumber + 1) MOD 14 + 1);
      ***}
      NewNumber := random(rand1, 1, 14);
```

```
    write('.');
    NewNumber := NewNumber + 56;              { put in range of 57 to 70...}
  end; { ...find an unused number... }

  if (check > 14) then
  begin
    writeln;
    write('ERROR: trouble in procedure ChoosePlanes, cannot find new number.')
    writeln;
  end;

  AlreadyChosen := AlreadyChosen + [NewNumber];

  IndexArray[(2 * i) + 1] := NewNumber;
  IndexArray[(2 * i) + 2] := NewNumber + 14;  { add corresponding distractor }
 end;

end; { ...of procedure ChoosePlanes... }
{-----------------------------------------}

PROCEDURE UpperCase;

VAR
      i              : integer;
      holder         : integer;
begin
 for i := 1 to length(name1) do
 begin
   holder := ord(name1[i]);
   if name1[i] in ['a'..'z'] then
     name1[1] := chr(holder - 32);
 end;
end { UpperCase };



FUNCTION Compare;

begin
 UpperCase(first);
 UpperCase(second);
 if (first = second) then Compare := true
   else Compare := false;
end;


PROCEDURE NewLines;

VAR
      i      : integer;

begin
 for i := 1 to count do writeln;
```

```
end { New Lines };


PROCEDURE ModWait;

VAR
      delay   : integer;
      i       : integer;

begin
  seed := abs(seed);
  delay := (seed mod 200);
  for i := 1 to delay do
  delay := delay;
end { of ModWait };



{*****************************}


Function ListLength;
 {
      This function finds and returns the length of the array parameter
    List, which is of type PicList.
 }

 VAR
      count   : integer;

 begin
  {writeln('Entering ListLength, list[1].name1 is ',List[1].Names[1]);}
  count := 1;

  while ((count <= MAXINDEX) and (List[count].Names[1] <> 'none123') and
                          (List[count].Names[1] <> 'NONE123') and
                          (List[count].Names[1] <> 'None123'))
  do
    begin
      {writeln('<',count,'>', List[count].Names[1]);}
      count := count + 1;
    end;

  {writeln('ListLength is ', count, '.');}
  ListLength := count - 1;
  {writeln('Leaving ListLength');
  readln;}
 end { ...of function ListLength... };
{-----------------------------------------}


{-----------------------------------------}
 Procedure BuildString;
 {
```

```
  This procedure will allow a string to be built character by character.
 A <backspace> will have the effect that it should... one character will
 be deleted off the end of the string.
  Each call to procedure BuildString will append one character ("NewChar")
 to the string "FinalString".
}

var
   strlen          : integer;
   StringEnd       : CharString;

begin
  StringEnd := ' ';
  strlen := length(FinalString);              { get current end of string }
  StringEnd[1] := NewChar;
  if (ord(NewChar) <> 8) then                          { not a backspace }
    begin
    if length(FinalString) < 20 then
    FinalString := concat(FinalString, StringEnd)
    end
  else                          { the character entered is a backspace... }
    if (strlen > 0)                      { ...and there is at least one
                                           character to get rid of... }
      then
      begin
        delete(FinalString, strlen, 1);
        write(' ',chr(8));
      end
    else write(' ');
end { ...of procedure BuildString...};
{-----------------------------------------}


{-----------------------------------------}
Function TimeRead;
{
  This function acts like a "readln", except that it also returns an
  integer which is the count of terak clock-ticks it took the user to
  enter the first two characters.  It uses the procedure BuildString.
}

VAR
      letter              : Char;
      KeysEntered         : integer;
      ElapsedTime         : integer;
      HighStart, LowStart : integer;     { high and low order starting
                                           clock values }
      HighStop, LowStop   : integer;

begin
  result := '';
  KeysEntered := 0;
  HighStart := 0;     LowStart := 0;
  HighStop := 0;      LowStop := 0;
```

```
  Time(HighStart, LowStart);
  while not EOLN do
  begin
    reset(keyboard);
    read(letter);

    KeysEntered := KeysEntered + 1;
    BuildString(result, letter);

    if ((KeysEntered = 2) or (EOLN and (KeysEntered < 2)))
      then Time(HighStop, LowStop);
  end; { of While loop for reading in characters }
  if KeysEntered > 0 then
    for x:= 1 to length(result) do if x <= length(result) then
      if result[x] = ' ' then delete(result,x,1);
  if result = '' then result := 'XXXXX';
  if ((LowStart > 0) and (LowStop <= 0)) then {if the clock counted to Max, the
                                        started negative...          }
    ElapsedTime := ((MaxInt - LowStart) + (MaxInt + LowStop)) else
    ElapsedTime := LowStop - LowStart;
  TimeRead := ElapsedTime;
 end { ...of function TimeRead... };
{-----------------------------------------}


{-----------------------------------------}
Procedure RemoveBlanks;
{
  This procedure will remove all trailing <space> and <return> characters
  from the end of string1.
}

VAR
      i             : integer;


begin
  while ((ord(string1[length(string1)])) in [32, 13])
  do                { if last character is a <space> or a <ret>...      }
  delete(string1, length(string1), 1);              { ...then remove it     }
end { ...of procedure RemoveBlanks... };
{-----------------------------------------}

Procedure Strip;

VAR
      i             : integer;

begin
  for i := length(string1) downto 1 do
  begin
    if NOT (string1[i] in ['a'..'z', 'A'..'Z', '0'..'9']) then
      delete(string1, i, 1);
  end;
```

```
end { ...of procedure Strip... };


{-----------------------------------------}
Function CheckAnswer;
{
 This function will compare the string "answer" with all possible correct
 answers found in NameRec and return "true" if a match is found, otherwise
 it will return "false". The differences between capital and small letters
 make no difference, for both "answer" and "Possibles" are converted to all
 capitals. Also, any non-alphanumeric characters in either name (such as
 "-", or "/", or space, will be stripped out before the comparison.
}

begin
 with Possibles do
 begin
  UpperCase(Names[1]);
  UpperCase(Names[2]);
  UpperCase(answer);

  Strip(Names[1]);
  Strip(Names[2]);
  Strip(answer);

  if (answer = Names[1]) or (answer = Names[2]) or
     (answer = concat(Names[1], Names[2])) or
     (answer = concat(Names[2], Names[1]))
  then
   CheckAnswer := true

  else
   CheckAnswer := false;

  if (answer = '') then CheckAnswer := False;


 end { ...of "with Possibles"... };
end { ...of function CheckAnswer... };
{-----------------------------------------}
END { of GameUn1 }.
```

```
;*****************************************************************************;
;                    FLASH IVAN INTRODUCTION                                  ;
;*****************************************************************************;
; NPRDC                DECEMBER 9, 1983            DAVID M.SETTER ;
;*****************************************************************************;
      .PROC  SOUND

; MACROS AND SYMBOLICS:
      .INCLUDE      SYMBOLICS.TEXT
      .INCLUDE      MACROS.TEXT
      .INCLUDE      SND_EFF.TEXT

; SAVE REGISTERS:
      PUSH   R5
      PUSH   R4
      PUSH   R3
      PUSH   R2
      PUSH   R1
      PUSH   R0


; TAKE CONTROL OF KB AND LP_EDB:
      BIC    #100,@#177560
      BIC    #100,@#177564


;*****************************************************************************;
; MAIN:                                                                       ;
;*****************************************************************************;
MAIN:
      CLR    R1
92$:
      MOV    #006,R5
      MOV    #20,R0
      ADD    #2,R1
      SUB    R1,R0
1$:
      PITCH  R0,R0,#12,#6
      SOB    R5,1$


      CMP    R1,#0
      BLE    92$




;*****************************************************************************;
; END OF MAIN.                                                                ;
;*****************************************************************************;
```

```
BIS     #100,@#177560
BIS     #100,@#177564


POP     R0
POP     R1
POP     R2
POP     R3
POP     R4
POP     R5

RTS     PC


.END
```

```
;*************************************************************************;
;                    FLASH IVAN INTRODUCTION                              ;
;*************************************************************************;
; NPRDC                 DECEMBER 9, 1983            DAVID M.SETTER ;
;*************************************************************************;
        .PROC   TIMEPI

; MACROS AND SYMBOLICS:
        .INCLUDE        MACROS.TEXT
        .INCLUDE        SYMBOLICS.TEXT
        .INCLUDE        SND_EFF.TEXT

; SAVE REGISTERS:
        PUSH    R5
        PUSH    R4
        PUSH    R3
        PUSH    R2
        PUSH    R1
        PUSH    R0


; TAKE CONTROL OF KB AND LP_EDB:
        BIC     #100,@#177560
        BIC     #100,@#177564


;*************************************************************************;
; MAIN:                                                                   ;
;*************************************************************************;
        MOV     #20,R0
MAIN:
        PITCH   #2,R0,#20,#2
        SOB     R0,MAIN



        MOV     #20,R1
55$:
        PITCH   #2,#1,#10,#2
        SOB     R1,55$









;*************************************************************************;
; END OF MAIN.                                                            ;
;*************************************************************************;
        BIS     #100,@#177560
```

```
        BIS     #100,@#177564

        POP     R0
        POP     R1
        POP     R2
        POP     R3
        POP     R4
        POP     R5

        RTS     PC

;.....
;.....
        .END
```

```
{$S+}
Program makestats;
CONST MAXINDEX = 100;
    MAXNAMES = 3;
    I_name = 'Newnames';
    numberpictures  = 89;          (* total # of pictures available.
                                   The number of planes actually
                                   used in a game is unimportant. *)
     name          = '#5:GAMES.DATA'; (* Disk file of played game stats *)
     across        = 12;            (* Formatting; # rows printed     *)
     maxgames      = 100;                          (*    per page.    *)
TYPE  STR15 = string[15];
    sstype = string[11];
    NEWREC = packed record
             names : packed array[1..MAXNAMES] of str15;
             block : integer;
             FULLSCREEN : BOOLEAN;
             TOPTHIRD   : BOOLEAN;
             BOTTHIRD   : BOOLEAN;
           end;
   ILIST = array[1..MAXINDEX] of NEWREC;
     nametype       = string[15];
     scale          = 0..100;
     gamestats      = record
      name          : nametype;          (* The player's name, and  *)
      ss            : sstype;
      date          : nametype;          (* the date of the game.   *)
      latency       : packed array[1..numberpictures] of integer; (* Statist
      confidence    : packed array[1..numberpictures] of scale;   (* for eve
      correct       : packed array[1..numberpictures] of boolean; (* plane f
     end; (* gamestats *)                                  (* each game. *)
VAR infodir : file of NEWREC;
   INFO_LIST : ILIST;
     total_lat,total_correct,total_conf:array[1..numberpictures] of real;
     lat_total,conf_total,correct_total : array[1..maxgames] of real;
     no_planes      : array[1..numberpictures] of integer;
     no_games       : array[1..maxgames] of integer;
     scoresfile     : file of gamestats;
     xx,x,y,letter,letterstep  : integer;
     outfile        : text;
     current        : gamestats;
     no_across      : integer;

procedure header(var outfile : text);
begin
   writeln(outfile,'PLAYER','PLANES':34);
   writeln(outfile,' #','------':38);
   for y := 1 to numberpictures do
     no_planes[y]     := 0;
   for y := 1 to maxgames do
     no_games[y]      := 0;
end;  (* of Procedure Header *)

procedure planenames(var outfile : text);
```

```
begin
  write(outfile,x,'.');
  if x < 10 then write(outfile,' ');
  if x < 29 then write(outfile,' (Top)   ')
  else if (x > 28) and (x < 57) then write(outfile,' (Side)  ')
  else write(outfile,' (Front) ');
   (*PLANE NAMES GO HERE*)
  write(outfile,INFO_LIST[x].names[1]:15,' ':20);
end;

procedure showconfidence(var outfile : text);
procedure showratings   (var outfile : text);
procedure showpeople    (var outfile : text);
begin
  write(outfile,' NAME          SS #         DATE ':40);
  writeln(outfile,'Average Confidence':22,'Percent Correct':17);
  reset(scoresfile,name);
  letterstep := 65;
  letter := 64;
  xx := 0;
  while not (eof(scoresfile)) do
    begin
      current :=scoresfile^;
      get(scoresfile);
      letter := letter + 1;
      if letter > 90 then
      begin
        letter := 65;
        letterstep := letterstep + 1;
      end;  (* of if *)
      xx := xx + 1;
      write(outfile,chr(letterstep),chr(letter),':',' ',current.NAME:15,
      current.SS:13,current.date:12);
      if not (no_planes[xx] = 0) then
        write(outfile,(conf_total[xx]/no_planes[xx]):12:1)
      else write(outfile,'-  ':12);
      if not (no_planes[xx] = 0) then
        writeln(outfile,round(100*correct_total[xx]/no_planes[xx]):17,'%')
      else writeln(outfile,'- ':18);
    end;(* of while *)
    close(scoresfile);
  writeln(outfile);
end;  (* of Procedure Showpeople *)
begin (* Procedure Showratings *)
  no_across := 1;
  repeat
    write(outfile,' ':4);
    for x := no_across to no_across + across - 1 do
    if x <= numberpictures then
      write(outfile,x:5,' ');
    writeln(outfile);
    write(outfile,' ':4);
    for x := no_across to no_across + across - 1 do
    if (x <= numberpictures) and (x < 10) then
```

```
    write(outfile,'-':5,' ')
  else if x <= numberpictures then
    write(outfile,'--':5,' ');
  writeln(outfile);
  reset(scoresfile,name);
  letterstep := 65;
  letter := 64;
  xx := 0;
  while not (eof(scoresfile)) do
  begin
    current :=scoresfile^;
    get(scoresfile);
    letter := letter + 1;
    xx := xx + 1;
    if letter > 90 then
    begin
      letter := 65;
      letterstep := letterstep + 1;
    end;  (* of if *)
    write(outfile,chr(letterstep),chr(letter),'   ');
    for y := no_across to x - 1 do
      if y <= numberpictures then
          if not (current.latency[y] = 0) then
          begin
            no_planes[xx] := no_planes[xx] + 1;
            no_games[y] := no_games[y] + 1;
            total_conf[y] := total_conf[y] + current.confidence[y];
            conf_total[xx] := conf_total[xx] + current.confidence[y];
            if current.correct[y] then
            begin
            total_correct[y] := total_correct[y] + 1;
            correct_total[xx] := correct_total[xx] + 1;
            write(outfile,'+':3)
            end (* of if *)
            else write(outfile,'-':3);
            write(outfile,round(current.confidence[y]):3);
          end (* of if *)
          else write(outfile,'   - ');
    writeln(outfile);
  end; (* of while *)
  close(scoresfile);
  writeln(outfile);
  for x :=1 to 80 do write(outfile,'-');
  writeln(outfile);
  no_across := no_across + across;
until no_across >= numberpictures;
writeln(outfile);
showpeople(outfile);
end; (* of Procedure Showratings *)
procedure showplanes(var outfile : text);
begin
  for x :=1 to 80 do write(outfile,'-');
  writeln(outfile);
  writeln(outfile,'PLANES:',' ':41,'Average Confidence','% Correct?':14);
```

```
  for x := 1 to numberpictures do
  begin
    planenames(outfile);
    if no_games[x] > 0 then write(outfile,total_conf[x]/no_games[x]:11:1)
    else write(outfile,'-' :11);
    if no_games[x] > 0 then
      write(outfile,round(100*total_correct[x]/no_games[x]):19,'%')
    else write(outfile,'- ' :20);
    writeln(outfile);
  end; (* of for *)
  for x :=1 to 80 do write(outfile,'-');
  writeln(outfile);
end;    (* of Procedure Showplanes *)

begin  (* Procedure Showconfidence *)
  for x:= 1 to numberpictures do
  begin
    total_correct[x] := 0;
    total_conf[x] := 0;
  end;
  for x:= 1 to maxgames do
  begin
    conf_total[x] := 0;
    correct_total[x] := 0;
  end;
  writeln(outfile);
  writeln(outfile,'RESPONSE CORRECTNESS and':48);
  writeln(outfile,'CONFIDENCE RATINGS':45);
  writeln(outfile,'  + = correct    ':45);
  writeln(outfile,'  - = wrong      ':45);
  for x :=1 to 80 do write(outfile,'-');
  writeln(outfile);
  header(outfile);
  showratings(outfile);
  showplanes(outfile);
  close(scoresfile);
end; (* of Procedure Showconfidence *)

procedure showlatency(var outfile : text);
procedure showratings(var outfile : text);
begin
  no_across := 1;
  repeat
    write(outfile,' ':4);
    for x := no_across to no_across + across - 1 do
    if x <= numberpictures then
      write(outfile,x:5,' ');
    writeln(outfile);
    write(outfile,' ':4);
    for x := no_across to no_across + across - 1 do
    if (x <= numberpictures) and (x < 10) then
      write(outfile,'-':5,' ')
    else if x <= numberpictures then
      write(outfile,'--':5,' ');
```

```
    writeln(outfile);
    reset(scoresfile,name);
    letterstep := 65;
    letter := 64;
    xx := 0;
    while not (eof(scoresfile)) do
    begin
      current :=scoresfile^;
      get(scoresfile);
      letter := letter + 1;
      xx := xx + 1;
      if letter > 90 then
      begin
        letter := 65;
        letterstep := letterstep + 1;
      end;  (* of if *)
      write(outfile,chr(letterstep),chr(letter),'  ');
      for y := no_across to x - 1 do
        if y <= numberpictures then
            if not (current.latency[y] = 0) then
            begin
              no_planes[xx] := no_planes[xx] + 1;
              no_games[y] := no_games[y] + 1;
              total_lat[y] := total_lat[y] + current.latency[y];
              lat_total[xx] := lat_total[xx] + current.latency[y];
              write(outfile,current.latency[y]:5,' ');
            end (* of if *)
            else write(outfile,'    - ');
      writeln(outfile);
    end; (* of while *)
    close(scoresfile);
    writeln(outfile);
    for x :=1 to 80 do write(outfile,'-');
    writeln(outfile);
    no_across := no_across + across;
  until no_across >= numberpictures;
  writeln(outfile);
end; (* of Procedure Showratings *)
procedure showplanes(var outfile : text);
begin
  for x :=1 to 80 do write(outfile,'-');
  writeln(outfile);
  writeln(outfile,'PLANES:',' ':41,'Average Latency');
  for x := 1 to numberpictures do
  begin
    planenames(outfile);
    if not (no_games[x] = 0) then write(outfile,round(total_lat[x]/no_games[x]):
    else write(outfile,'-':9);
    writeln(outfile);
  end; (* of for *)
  for x :=1 to 80 do write(outfile,'-');
  writeln(outfile);
end; (* of Procedure Showplanes *)
procedure showpeople(var outfile : text);
```

```
begin
  write(outfile,' NAME          SS #          DATE ':40);
  writeln(outfile,'Average Latency':22);
  reset(scoresfile,name);
  letterstep := 65;
  letter := 64;
  xx := 0;
  while not (eof(scoresfile)) do
    begin
      current :=scoresfile^;
      get(scoresfile);
      letter := letter + 1;
      if letter > 90 then
      begin
        letter := 65;
        letterstep := letterstep + 1;
      end;  (* of if *)
      xx := xx + 1;
      write(outfile,chr(letterstep),chr(letter),':',' ',current.NAME:15,
      current.SS:12,current.date:12);
      if not (no_planes[xx] = 0) then
        writeln(outfile,round(lat_total[xx]/no_planes[xx]):12,'  ')
      else writeln(outfile,'-  ':14);
    end;(* of while *)
    close(scoresfile);
  writeln(outfile);
end; (*of Procedure Showpeople *)

begin  (* Procedure Showlatency *)
  for x := 1 to maxgames do lat_total[x] := 0;
  writeln(outfile);
  writeln(outfile,'LATENCIES (in milliseconds)':50);
  for x :=1 to 80 do write(outfile,'-');
  writeln(outfile);
    header(outfile);
    showratings(outfile);
    showpeople(outfile);
    writeln(outfile);
  showplanes(outfile);
  reset(scoresfile,name);
  close (scoresfile,purge);  (* ,purge *)
end; (* of Procedure Showlatency *)

procedure FROMDISK;
var H : integer;
begin
    reset(infodir,I_name);
    FOR H := 1 TO MAXINDEX DO BEGIN
      Info_List[H] {** II_list[H] **}  := infodir^;
      if not EOF(infodir) then get(infodir);
    end;
    close(infodir);
end;
```

```
begin   (* MAIN *)
 for x := 1 to numberpictures do
  begin
    total_correct[x] := 0;
    total_conf[x]    := 0;
    total_lat[x]     := 0;
  end; (* of for *)
  FROMDISK;
  rewrite(outfile,'CONFIDENCE.TEXT');
  showconfidence(outfile);
  close(outfile,lock);
  rewrite(outfile,'LATENCY.TEXT');
  showlatency(outfile);
  close(outfile,lock);
   rewrite(scoresfile,name);
     close(scoresfile,lock);
end. (* MAIN *)
```

```
Program Driver;
CONST numberpictures  = 85;              (* total # of pictures available.
                                         The number of planes actually
                                         used in a game is unimportant. *)
      name            = '#5:GAMES.DATA'; (* Disk file of played game stats
TYPE  nametype        = string[15];
      sstype          = string[11];
      scale           = 0..100;
      gamestats       = record
        name          :nametype;
        SS            : sstype;              (* The player's name, and  *)
        date          : nametype;              (* the date of the game.   *)
        latency       : packed array[1..numberpictures] of integer; (* Statist
        confidence    : packed array[1..numberpictures] of scale;   (* for eve
        correct       : packed array[1..numberpictures] of boolean; (* plane f
      end; (* gamestats *)                                          (* each game. *)
VAR current           : file of gamestats;

begin   (* MAIN *)
  rewrite(current,name);
  close(current,lock);
end. (* MAIN *)
```

```
{$S+}
PROGRAM MAKEDIR2;

USES MENUS;

CONST I_NAME = '#5:NEWNAMES';
    MAXINDEX = 100;
    MAXNAMES = 3;
    menu1X = 4;
    menu1Y = 6;
    s1 = 'Name 1';
    s2 = 'Name 2';
    s3 = 'Fotofile name:';
    s4 = 'Fullscreen [T/F]';
    s5 = 'Top Third  [T/F]';
    s6 = 'Mid Third  [T/F]';
    s7 = 'Bot Third  [T/F]';

    MAXDIR = 77;      (*MAX NUMBER OF ENTRIES IN A DIRECTORY*)
    VIDLENG = 7;      (*NUMBER OF CHARS IN A VOLUME ID*)
    TIDLENG = 15;     (*NUMBER OF CHARS IN TITLE ID*)
    FBLKSIZE = 512;   (*STANDARD DISK BLOCK LENGTH*)
    DIRBLK = 2;       (*DISK ADDR OF DIRECTORY*)
    NAME_LEN = 23;    {Length of CONCAT(VIDLENG,':',TIDLENG)}

TYPE
    DATEREC = PACKED RECORD
            MONTH: 0..12;         (*0 IMPLIES DATE NOT MEANINGFUL
            DAY: 0..31;          (*DAY OF MONTH*)
            YEAR: 0..100          (*100 IS TEMP DISK FLAG*)
          END (*DATEREC*) ;

                          (*VOLUME TABLES*)
    VID = STRING[VIDLENG];

                          (*DISK DIRECTORIES*)
    DIRRANGE = 0..MAXDIR;
    TID = STRING[TIDLENG];

    FILEKIND = (UNTYPEDFILE,XDSKFILE,CODEFILE,TEXTFILE,
          INFOFILE,DATAFILE,GRAFFILE,FOTOFILE,SECUREDIR);

    DIRENTRY = PACKED RECORD
            DFIRSTBLK: INTEGER;   (*FIRST PHYSICAL DISK ADDR*)
            DLASTBLK: INTEGER;    (*POINTS AT BLOCK FOLLOWING*)
            CASE DFKIND: FILEKIND OF
             SECUREDIR,
             UNTYPEDFILE: (*ONLY IN DIR[0]...VOLUME INFO*)
               (FILLER1 : 0..2048; {for downward compatibility,13 bits}
                DVID: VID;              (*NAME OF DISK VOLUME*)
                DEOVBLK: INTEGER;       (*LASTBLK OF VOLUME*)
                DNUMFILES: DIRRANGE;    (*NUM FILES IN DIR*)
                DLOADTIME: INTEGER;     (*TIME OF LAST ACCESS*)
                DLASTBOOT: DATEREC);    (*MOST RECENT DATE SETTING*)
```

```
                XDSKFILE,CODEFILE,TEXTFILE,INFOFILE,
                DATAFILE,GRAFFILE,FOTOFILE:
                  (FILLER2 : 0..1024; {for downward compatibility}
                   STATUS : BOOLEAN;        {for FILER wildcards}
                   DTID: TID;              (*TITLE OF FILE*)
                   DLASTBYTE: 1..FBLKSIZE; (*NUM BYTES IN LAST BLOCK*)
                   DACCESS: DATEREC)       (*LAST MODIFICATION DATE*)
              END (*DIRENTRY*) ;


    Directory = ARRAY[DIRRANGE] OF DIRENTRY;

    STR15 = STRING[15];
    INFO_REC = PACKED RECORD
                 NAMES: PACKED ARRAY[1..MAXNAMES] OF STR15;
                 BLOCK : INTEGER;
                 FULLSCREEN : BOOLEAN;
                 TOPTHIRD   : BOOLEAN;
                 MIDTHIRD   : BOOLEAN;
                 BOTTHIRD   : BOOLEAN;
               END;
     ILIST = ARRAY[1..MAXINDEX] OF INFO_REC;

VAR INFODIR : FILE OF INFO_REC;
   INFO_LIST : ILIST;
   CH : CHAR;
   I,J : integer;
   noloop,loopit,done : boolean;
   choice : integer;
   fin,ans,response : char;
   MENU1 : MENURECORD;   { from library program MENUS }
   dir  : Directory;
   LISTFILE : TEXT;      { used in QUICKLIST to output dir to textfile }


(*****************************************************************************)

procedure FROMDISK(VAR II_LIST . ILIST);
var H : integer;
begin
   reset(infodir,I_NAME);
   H := 1;
   while not EOF(infodir) do begin
     II_list[H] := infodir^;
     H := H + 1;
     get(infodir);
   end;
   close(infodir);
end;


procedure TOODISK(VAR II_LIST : ILIST);
var Z : integer;
begin
```

```
  rewrite(infodir,I_NAME);
  for Z := 1 to MAXINDEX do
  begin
    infodir^ := II_list[Z];
    put(infodir);
  end;
  close(infodir,lock);
end;


(*****************************************************************************)


  PROCEDURE CLEARSPACE( D : INTEGER);
  VAR E : INTEGER;
  BEGIN
    FOR E := 1 TO D DO
      WRITE(CHR(32));
    FOR E := 1 TO D DO
      WRITE(CHR(8));
  END;

  procedure ClearLine;
  begin
    write('
    for i := 1 to 67 do write(chr(8));
  end;


  procedure Boolwrite(A:boolean);
  begin
    if A=TRUE then write('TRUE')
            else write('FALSE');
  end;


  procedure Boolread(VAR A:boolean);
  var ch : char;
  begin
    read(ch);
    if ((ch='T') or (ch='t')) then A:=TRUE
                              else A:=FALSE;
  end;

  procedure ShowValue(p:integer);
  begin

  with Info_list[J] do begin
    if p=1 then begin
      ClearLine;
      gotoxy(menu1X + 26, menu1Y + P + 1);
      ClearSpace(15);
      write(names[1]);
    end;
```

```
  if p=2 then begin
    ClearLine;
    gotoxy(menu1X + 26, menu1Y + P + 1);
    ClearSpace(15);
    write(names[2]);
    end;
  if p=3 then begin
    ClearLine;
    gotoxy(menu1X + 26, menu1Y + P + 1);
    ClearSpace(15);
    write(names[3]);
  end;
  if p=4 then begin
    ClearLine;
    gotoxy(menu1X + 26, menu1Y + P + 1);
    ClearSpace(5);
    boolwrite(fullscreen);
  end;
  if p=5 then begin
    ClearLine;
    gotoxy(menu1X + 26, menu1Y + p + 1);
    ClearSpace(5);
    boolwrite(topthird);
  end;
  if p=6 then begin
    ClearLine;
    gotoxy(menu1X + 26, menu1Y + P + 1);
    ClearSpace(5);
    boolwrite(midthird);
  end;
  if p=7 then begin
    ClearLine;
    gotoxy(menu1X + 26, menu1Y + P + 1);
    ClearSpace(5);
    boolwrite(botthird);
  end
 end {with}
end;


procedure MakeMenu;
var title,convert:string;
begin
  { str(J,convert); }
  { title := concat('INDEX ',convert); }
  MenuNew(menu1,  menu1X, menu1Y, 20, 7,'INDEX NUMBER');
  MenuInsert(menu1, s1, menu1.len+1);
  MenuInsert(menu1, s2, menu1.len+1);
  MenuInsert(menu1, s3, menu1.len+1);
  MenuInsert(menu1, s4, menu1.len+1);
  MenuInsert(menu1, s5, menu1.len+1);
  MenuInsert(menu1, s6, menu1.len+1);
  MenuInsert(menu1, s7, menu1.len+1);
end;
```

```
procedure DoMenu( TheMenu : MenuRecord; VAR choice : integer);
var last : char;
begin
  gotoxy(0, 2);
  writeln('Use the arrow keys to move among the choices, ');
  writeln('type "S" to select which item to change. ');
  MenuDisplay(TheMenu);
  last := MenuUserSel(TheMenu);
  choice := TheMenu.curItem;
end;



procedure NewOne;
var iter : integer;
    response,bool : char;
    done : boolean;

  procedure NameCase(VAR name:str15; st:string; c:integer);
  begin
    gotoxy(0,menu1Y + 14);
    ClearLine;
    write(st,' is currently ');
    writeln('"',name,'" ');
    ClearLine;
    write('Enter the new ',st,': ---> ');
    reset(input);
    readln(name);
  (*UpperCase(name);*)    { from SceneUn1 }
    WRITELN;
    CLEARLINE;
  ShowValue(c);
    gotoxy(0,menu1Y + 14);
    ClearLine;
    gotoxy(0,menu1Y + 15);
    ClearLine;
end;

procedure BoolCase(c:integer);
var i : integer;
begin
  with Info_list[J] do begin
    fullscreen := false;
    topthird := false;
    midthird := false;
    botthird := false;
    if c = 4 then fullscreen := true
      else if c = 5 then topthird := true
        else if c = 6 then midthird := true
          else if c = 7 then botthird := true;
    for i := 4 to 7 do ShowValue(i);
```

```
    gotoxy(0,menu1Y + 14);
    writeln('For boolean fields, selection automatically sets');
    write('  selected field TRUE, other fields FALSE.  Hit <RET>.');
    readln;
    gotoxy(0,menu1Y + 14);
    ClearLine;
    gotoxy(0,menu1Y + 15);
    ClearLine;
  end; { with }
end;


begin  { NewOne }
 MakeMenu;
 PAGE(OUTPUT);
 gotoxy(20,6);      { Writes index number to screen }
 ClearSpace(3);
 write(J);
 for iter := 1 to 7 do ShowValue(iter);
 done := false;
 while (not done) do
  begin
   MenuReset(menu1);
   DoMenu(menu1, choice);
   gotoxy(20,6);      { Writes index number to screen }
   ClearSpace(3);
   write(J);
   if choice=1
     then NameCase(Info_list[J].names[1],s1,choice)
   else if choice=2
     then NameCase(Info_list[J].names[2],s2,choice)
   else if choice=3
     then NameCase(Info_list[J].names[3],s3,choice)
   else BoolCase(choice);
   gotoxy(menu1X + 36, menu1Y + 8);
   write('        Change more values? [Y/N] ');
   read(keyboard,response);
   if (response in ['N', 'n']) then done:=true else done:=false;
  end {while}
end;



procedure procA;
var loopit : boolean;
   fin : char;
begin
    repeat
     repeat
       loopit := false;
       gotoxy(1,22);
       write('Edit Index Number: --> ');
       read(J);                          { J GLOBAL TO MAKEDIR }
       if (J<1) OR (J>MAXINDEX) then begin
```

```
          loopit:=true;
          gotoxy(1,22);
          writeln('Value out of range. Type <RET> to continue');
          readln;
        end;
        gotoxy(0,22);
        CLEARLINE;
      until not loopit;
      NewOne;
      menudisplay(menu1);
      gotoxy(1,22);
      write('Edit Another? [Y/N] --> ');
      read(fin);
      gotoxy(0,22);
      CLEARLINE;
    until (fin in ['N', 'n']);
end; { procA }


procedure procB;
var lower,higher:integer;
   again : boolean;
begin
 repeat
   again := false;
   gotoxy(0,0);
   ClearLine;
   write('Enter lower index bound space, upper index bound: --> ');
   read(lower,higher);
   if (lower<1) or (lower> MAXINDEX) or (higher<1) or (higher>MAXINDEX)
     then again:=true
       else
         if (lower>higher) then again:=true;
 until not again;
 for J:= lower to higher do    { J is global to MAKEDIR }
   begin
     NewOne;
     menudisplay(menu1);
   end;
end;   { procB }

(*****************************************************************)

PROCEDURE CHANGER;
VAR i : integer;

 FUNCTION findnum(VAR alias : string) : INTEGER;
 VAR n : integer;
    found,done : boolean;
 BEGIN
   done := FALSE;
   found := FALSE;
   n := 1;
   REPEAT
```

```
    IF dir[n].dtid=alias THEN BEGIN
      findnum := dir[n].dfirstblk;
      done := TRUE;
      found := TRUE;
    END
      else n := n+1;
  UNTIL ((n=MAXDIR+1) or done);
  IF found=FALSE
    THEN WRITELN('Can not find ',alias,' on disk in upper drive (#5) ');
 END;

BEGIN  { CHANGER }
 PAGE(OUTPUT);
 WRITELN('CONVERTING ... ');
 UNITREAD(5,dir,SIZEOF(dir),DIRBLK);
 FOR I:= 1 TO MAXINDEX DO
    INFO_LIST[I].block := findnum(INFO_LIST[I].names[3]);
 TOODISK(INFO_LIST);
 WRITELN('DIRECTORY CONVERTED. ');
END;

 PROCEDURE INITDIR;
 begin
  for J := 1 to MAXINDEX do begin
    Info_list[J].names[1] := 'none123';
    Info_list[J].names[2] := 'none123';
    Info_list[J].names[3] := 'FLAGS.FOTO';
    Info_list[J].block := 0;
    Info_list[J].fullscreen := TRUE;
    Info_list[J].topthird := FALSE;
    Info_list[J].midthird := FALSE;
    Info_list[J].botthird := FALSE;
  end;
  Info_list[90].names[3] := 'EAGLE1.FOTO';   { initializing system records }
  Info_list[91].names[3] := 'EAGLE2.FOTO';

  Info_list[92].names[1] := '';
  Info_list[92].names[2] := 'ADM1';
  Info_list[92].names[3] := 'EX3.FOTO';

  Info_list[93].names[1] := '';
  Info_list[93].names[2] := 'SPACE SHUTTLE';
  Info_list[93].names[3] := 'EX4.FOTO';

  Info_list[94].names[1] := '';
  Info_list[94].names[2] := 'XFV12';
  Info_list[94].names[3] := 'EX1.FOTO';
  Info_list[94].fullscreen := false;
  Info_list[94].topthird := true;

  Info_list[95].names[1] := '';
  Info_list[95].names[2] := 'X29';
  Info_list[95].names[3] := 'EX5.FOTO';
```

```
    Info_list[96].names[1] := 'FIGHTING FALCON';
    Info_list[96].names[2] := 'F16';
    Info_list[96].names[3] := 'EX1.FOTO';
    Info_list[96].fullscreen := false;
    Info_list[96].midthird := true;

    Info_list[97].names[1] := '';
    Info_list[97].names[2] := 'SPACE SHUTTLE';
    Info_list[97].names[3] := 'EX2.FOTO';

    Info_list[98].names[3] := 'IN1.FOTO';
    Info_list[99].names[3] := 'IN2.FOTO';
  END;

PROCEDURE EDITDIR;
var ans : char;
Begin
  MenuInit;
  MenuVars^.SelChars := MenuVars^.SelChars + ['S', 's'];
  MenuVars^.EscChars := MenuVars^.EscChars + ['S', 's'];
  PAGE(OUTPUT);
  REPEAT
    gotoxy(0,0);
    write('Edit OLD "NEWNAMES" Directory, or make NEW "NEWNAMES" Directory? [O/N
    read(keyboard,ans);
  UNTIL (ans in ['o', 'O', 'n', 'N']);
  if (ans in ['o', 'O']) then
    FROMDISK(Info_list)
  else INITDIR;
  PAGE(OUTPUT);
  REPEAT
    WRITELN('EDITING OPTIONS: ');
    WRITELN('      1 : INDEX CHOICE ');
    WRITELN('      2 : AUTO-INDEX ');
    WRITELN;
    WRITE('Type <1> or <2>  -> ');
    READ(ans);
  UNTIL (ans in ['1', '2']);
  PAGE(OUTPUT);
  IF ans = '2' then procB
    ELSE procA;
  CHANGER;
End;  { MakeDir }


procedure LISTER;
VAR i : integer;
    ans,ch : char;

  procedure Fileboolwrite(A:boolean);
  begin
    if A=TRUE then writeln(LISTFILE,'TRUE')
            else writeln(LISTFILE,'FALSE');
  end;
```

```
BEGIN
 PAGE(OUTPUT);
 WRITELN('QUICKLIST'); WRITELN;
 FROMDISK(INFO_LIST);
 FOR i := 1 to MAXINDEX do begin
  With INFO_LIST[i] do begin
   Writeln('Index #:  ',i);
   Writeln('NAMES[1]: ', names[1]);
   Writeln('NAMES[2]: ', names[2]);
   Writeln('NAMES[3]: ', names[3]);
   Writeln('BLOCK: ',block);
   Write(s4,' :   '); boolwrite(fullscreen); writeln;
   Write(s5,' :   '); boolwrite(topthird); writeln;
   Write(s6,' :   '); boolwrite(midthird); writeln;
   Write(s7,' :   '); boolwrite(botthird); writeln;
   Readln;
  End;  { with }
 End;   { for }
 Page(output);
 writeln('Do you want a listing sent to QUICKLIST.TEXT on the bottom disk?');
 read(keyboard,ans);
 if (ans in ['y','Y']) then begin
  REWRITE(LISTFILE,'QUICKLIST.TEXT');   { QUICKLIST.TEXT is output file }
  FOR i := 1 to MAXINDEX do begin
   With INFO_LIST[i] do begin
    Writeln(LISTFILE,'Index #:  ',i);
    Writeln(LISTFILE,'NAMES[1]: ', names[1]);
    Writeln(LISTFILE,'NAMES[2]: ', names[2]);
    Writeln(LISTFILE,'NAMES[3]: ', names[3]);
    Writeln(LISTFILE,'BLOCK: ',block);
    Write(LISTFILE,s4,' :   '); Fileboolwrite(fullscreen);
    Write(LISTFILE,s5,' :   '); Fileboolwrite(topthird);
    Write(LISTFILE,s6,' :   '); Fileboolwrite(midthird);
    Write(LISTFILE,s7,' :   '); Fileboolwrite(botthird);
    Writeln(LISTFILE);
   End;  { with }
  End;   { for }
  CLOSE(LISTFILE,LOCK);
 end; { if }
END;

PROCEDURE BUGS;
TYPE SCORES_REC  = RECORD
            GAMENAME : STRING[15];
            SCORE    : INTEGER;
           END;
VAR  HILIST  :  ARRAY[1..10] OF SCORES_REC;
    HISCOREFILE : FILE OF SCORES_REC;
    I,J,H : INTEGER;
    CH : CHAR;
    TEMPNAME : STRING[15];
    TEMPSCORE : INTEGER;
BEGIN
 PAGE(OUTPUT);
```

```
  WRITELN('THIS PROGRAM CREATES A FILE CALLED HISCORE.DATA');
  WRITELN('  WHICH IS PUT ON THE FOTOFILE DISK IN THE UPPER DRIVE');
  WRITELN;
  WRITELN('  ENTER THE TOP TEN SCORES WITH CORRESPONDING NAMES:');
  WRITELN;
  FOR H:=1 TO 10 DO BEGIN
   WRITELN('NUMBER ',H);
   WRITELN('ENTER NAME --> '); READLN(HILIST[H].GAMENAME);
   WRITELN('ENTER SCORE -> '); READLN(HILIST[H].SCORE);
   WRITELN;
  END;
  FOR J := 9 DOWNTO 1 DO BEGIN
   FOR I := 1 TO J DO BEGIN
    IF HILIST[I].SCORE < HILIST[I+1].SCORE THEN
     BEGIN
      TEMPNAME := HILIST[I].GAMENAME;
      TEMPSCORE := HILIST[I].SCORE;
      HILIST[I].GAMENAME := HILIST[I+1].GAMENAME;
      HILIST[I].SCORE := HILIST[I+1].SCORE;
      HILIST[I+1].GAMENAME := TEMPNAME;
      HILIST[I+1].SCORE := TEMPSCORE;
     END;
   END;
  END;
  REWRITE(HISCOREFILE,'#5:HISCORE.DATA');
  FOR H:= 1 TO 10 DO BEGIN
   HISCOREFILE^ := HILIST[H];
   PUT(HISCOREFILE);
  END;
  CLOSE(HISCOREFILE,LOCK);
END;


(************************** MAIN ***************************************)

BEGIN
  NOLOOP := FALSE;
  REPEAT
   PAGE(OUTPUT);
   WRITELN('************************** MAKEGAME PROGRAM **********************
   REPEAT
    WRITELN;
    WRITELN('NOTE: Be sure disk with game FOTOFILES is in upper disk drive. ')
    WRITELN;
    WRITELN('MAKEGAME OPTIONS: ');
    WRITELN('        1 : EDIT DIRECTORY ');
    WRITELN('        2 : CONVERT DIRECTORY ');
    WRITELN('        3 : QUICKLIST      ');
    WRITELN('        4 : MAKE HISCORES FILE ');
    WRITELN('        5 : QUIT');
    WRITELN;
    WRITE('Type <1> <2> <3> <4> or <5> --> ');
    READ(keyboard,ch);
   UNTIL (ch in ['1','2','3','4','5']);
```

```
  IF ch='2' THEN BEGIN
    PAGE(OUTPUT);
    FROMDISK(INFO_LIST);
    CHANGER;
  END
  ELSE IF CH='1' THEN EDITDIR
    ELSE IF CH='3' THEN LISTER
      ELSE IF CH='4' THEN BUGS
        ELSE IF CH='5' THEN NOLOOP := TRUE;
 UNTIL NOLOOP;
END.
```